Durra: A Task-level Description Language

Mario R. Barbacci[1,2] and Jeannette M. Wing[2]
Carnegie Mellon University
Pittsburgh, PA 15213, U.S.A.

Abstract

Computation-intensive, real-time applications such as vision, robotics, and vehicular control require efficient concurrent execution of multiple *tasks*, e.g., sensor data collection, obstacle recognition, and global path planning, devoted to specific pieces of the application. At CMU we are developing some of these applications and the hardware and software environments to support them, and in this paper we present a new language, Durra, to write what we call *task-level application descriptions*. Although the language was developed with a concrete set of needs, we aim at a broader class of applications and hardware implementations. After a brief description of the nature of these applications and a scenario for the development process, we concentrate on the language and its main features.

1 Introduction

We are interested in a class of real-time, embedded applications in which a number of concurrent, large-grained tasks cooperate to process data obtained from physical sensors, make decisions based on these data, and send commands to control motors and other physical devices. Since the speed and resources required of each task may vary, these applications can best exploit a computing environment consisting of multiple special- and general-purpose, loosely connected processors. 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 also 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 and the intermediate queues required to store the data as it moves from producer to consumer processes. A *task-level description language* is a notation for writing these application descriptions.

This paper addresses the design of Durra, a *task-level description language* [2]. 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 in some kind of executable "machine language." Rather, it is to be understood as a description of the structure and behavior of a logical machine, to 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.

2 Scenario

We assume that each of the processors in a heterogeneous machine has languages, compilers, libraries of programs, and other software development tools that cater to the special properties of a processor's architecture. For example, if an application requires a task for matrix multiplications, we assume code for it exists on one or more processors in the heterogeneous machine, perhaps in assembly language on an array processor or in C on a workstation.

Developing programs for some of the more exotic machines involves the selection of algorithms appropriate to a machine's architecture and the painstaking testing and tuning of the code to take advantage of any special features of the machine. This is a slow and difficult process and it would be natural to try to reuse these programs in multiple applications.

---

[1]Software Engineering Institute, [2]Department of Computer Science

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.

We assume therefore the existence of libraries of reusable programs, that is, programs that can be shared across applications (e.g., "feature detection" routines that could be shared in a variety of vision applications). The programs may be written in different programming languages (e.g., Ada, C, Common Lisp, as well as assembly language or even microcode), and executable on the different processors. We also assume a run time system that follows a scheduler's directives for downloading and executing the programs.

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

These three phases are illustrated in Figure 1. During the first phase, the developer breaks the application into specific tasks (e.g., sensor processing, feature recognition, map database management, route planning, etc.) and writes code implementing the tasks. For a given task, there may be many implementations, differing in programming language (e.g., one written in C or one written in assembly language), processor type (e.g., Motorola 68020 or IBM 1401), performance characteristics, or other attributes.

For each implementation of a task, the developer writes a *task description* and enters it into the *library*. This is where the task-description language first enters the picture. The user writes specifications of each task's performance and functionality, the types of data it produces or consumes, the ports it uses to communicate with other tasks, and other attributes of the task.

During the second phase 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. When the application description is compiled, the compiler generates a set of resource allocation and scheduling commands to be interpreted by the *scheduler*.

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

This scenario is not restricted to a static configuration of processes. The language allows the specification of alternative configurations and the conditions under which a configuration is selected. The dynamic reconfiguration of the process-queue graph is, of course, the responsibility of the scheduler. In the remainder of the paper, we concentrate on the language aspects of the above scenario, in particular, the writing of task descriptions and task selections.

3 Task Descriptions

Task descriptions, written and stored in task libraries, are building blocks for task-level programs. A task description contains the essential information that an application programmer needs to build task-level applications descriptions. This information is provided in several clearly identifiable clauses of a task description (see Figure 2): (1) its interface to other tasks (**ports**) and to the scheduler (**signals**), (2) its **attributes**, (3) its functional and timing **behavior**, and (4) its internal **structure**, thereby allowing for hierarchical task descriptions. See the Durra Reference Manual [2] for the precise syntax and further explanation of the semantics.

Perhaps the most interesting features are the *behavioral* information and the *structural* information. Other features, such as task, queue, and port names, input and output data types flowing in queues and through ports, and exceptional conditions signalled or handled, are not unlike similar constructs in conventional programming languages. The *attribute* information captures a number of additional properties of the task that are not easily cast in terms of the other pieces.

3.1 Interface Information

Interface information defines the ports of the processes instantiated from the task and the signals used by these processes to communicate with the scheduler. Here is a concrete example:

```
ports
    in1: in heads;
    out1, out2: out tails;
signals
    stop, start, resume: in;
    range_error, format_error: out;
```

A port declaration specifies the direction and the type of the data moving through the port. An **in** port takes input data from a queue, an **out** port deposits data into a queue. A signal declaration specifies only the direction of the scheduler messages. 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.

The data types specified in port declarations are declared independently of the tasks and are also stored in the library. In our language, these data type declarations specify scalars (of possible variable length), arrays of types, or even unions of other types, as shown in the following 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
```

3.2 Attribute Information

Attribute information specifies 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 selection (to be defined in section 4), 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.

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.

```
attributes
   author = "jmw";
   implementation = "/usr/jmw/sample.o";
   Queue_Size = 25;
```

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 example above, 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. 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 and the rules for matching of attributes are attribute dependent.

3.3 Behavioral Information

Behavioral information specifies functional and timing properties about the task. Durra uses standard axiomatic pre- and post-conditions to describe functionality and extended path expressions to describe timing. 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 through 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 based on first-order logic. The pre- and post-conditions constitute a simple Larch interface specification [4, 5]. The Larch Shared Language is used as the assertion language in these predicates. The formal meaning of the combined functional and timing behavior is defined using Jahanian and Mok's Real-Time Logic [6]. In the following example we illustrate the nature of the behavioral information without getting into details about their formal meaning; we encourage the reader to see [3] for the full details.

Consider a matrix multiplication task (Figure 3) that takes input matrices from two input queues and places 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.

The **timing** clause states that the task does not start executing until both input queues contain data. Once that condition is satisfied, the task will remove its input data from both input queues concurrently (the Dequeue operations), will operate on the data for between 10 and 15 seconds (this "computation" time is lumped together under the "delay" operation), and finally will deposit some output in the output queue. The **when** condition places a constraint on the state of the queues (not on the state of the data in the queues).

3.4 Structural Information

Structural information defines a process-queue graph (see, for example, Figure 4) and possible dynamic reconfiguration of the graph. Three kinds of declarations and one kind of statement can appear as structural information. This is illustrated in Figure 5, which shows the Durra (i.e., textual) version of the structured task of the same name in Figure 4.

A process declaration of the form *process_name*: **task** *task_selection* creates a process as an instance of the specified task. Since a given task (e.g., convolution) might have a number of different implementations that differ along different dimensions such as algorithm used, code version, performance, processor type, etc., the task selection in a process declaration specifies the desirable features of a suitable implementation. The presence of task selections within task descriptions provides direct linguistic support for reflecting hierarchically structured tasks (see Section 4.)

A queue declaration such as *queue_name queue_size*: *port_name_1* > *data_transformation* > *port_name_2* creates a queue through which data flows from an output port of a process (port_name_1) into the input port of another process (port_name_2). Port names must be unique within a task description. Outside their task (e.g., in a queue declaration) 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".

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 needed if the 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:

```
queue
   q1: p1 > > p2 ;
   q2: p1 > (2 1) transpose > p2 ;
   q3[100]: p1 > xyz > p2 ;
```

In the first example two ports, p1 and p2, are connected through an unbounded queue, q1, such that data flows from p1 to p2. The two ports must have the same type and no data transformations are performed. In the second example the data items (arrays) are transposed while in the queue. In the third example, the two ports are connected through a bounded (size=100) queue and the data items are transformed in the queue by a process "xyz".

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.

A reconfiguration statement of the form

```
if condition then
     remove   process-names
     process  process-declarations
     queues   queue-declarations
end if;
```

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 replaced 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.

4 Task Selections

As illustrated in the previous section, a process is an instance of a task specified in the process declaration. Given that a number of alternative task implementations might exist in the library, it is necessary to specify in the process declaration the desirable properties of the appropriate implementation. Here are some examples of process declarations, which in turn are used to select tasks:

```
process
  p1: task obstacle_finder;
  p2: task obstacle_finder
        ports foo: in heads, bar: out tails;
      end obstacle_finder;
  p3: task obstacle_finder
        attributes author="mrb";
      end obstacle_finder;
```

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. The task selection contains at least the name of a task and, optionally, interface, attribute, and behavior requirements (i.e., anything but structural information), and is used to select among a number of alternative task implementations.

A task can therefore be identified and selected from the library just by its name (if the name is unique in the library), by its interface properties (e.g., port types), by its attributes (e.g., version number), by its functional or timing behavior (e.g., a pre-condition), or by any combination of all of these.

For example, assume a declaration of a process, p, that includes the following task specification:

```
process
   p: task t
      attributes
         author = "jmw"; version = "45";
      end t;
```

The library search will proceed as follows. First, the task name, "t", is used to select as candidates all library task descriptions with the same name. Next, the attribute "author" in the task selection specifies the value "jmw" and this further reduces the set of candidates. Finally, the attribute "version" in the task selection specifies the value "45" and this reduces even further the remaining set of candidates. Since no additional information is given in the task selection, the candidates left uniquely identify those task implementations that could be used to implement the process at run time. Obviously, a task selection could be too constraining, eliminating all possible candidates or it could be too unconstraining, yielding more that one

possible matching task description (and, by implication, more than one task implementation). In the former case, an error is reported by the compiler. In the latter case, a random choice is made.

The rules for matching task selections with task descriptions vary depending on the construct being tested. Thus, for matching port types, a simple name equality is required. For matching attributes, the user can specify (in the task selection) conjunctions and disjunctions of attribute values (e.g., **author** = "mrb" **or** "jmw";). Finally, for matching behavior, the behavioral information of a candidate task description in the library must imply that of the task selection. This task selection mechanism provides flexible support for the reusability of code (task implementations) across applications although, we hasten to add, this feature is still untried and is likely to change based on real use.

5 Status and Conclusions

Our original motivation for designing and implementing a task-level language was to fill a need of two communities:

- Application programmers who want to exploit the capabilities of a computing environment that includes not only standard general-purpose processors and workstations but also high-speed special-purpose multiprocessors, all of which are networked together.
- Hardware designers who provide this broad range of computing capabilities and need customers to use their new configurations as different processors and communication links (e.g., optical switches) became available.

What was missing was a high-level *language* usable by the application programmers but targetable for the possibly changing hardware configurations. The language should let users focus their attention on describing their application at a *task* level rather than at a process or procedural level, without losing the ability to exploit the special features of each processor. We were furthermore constrained by the fact that enough "low-level" software, e.g., C and assembly programs that do number-intensive image processing, had been developed by both communities such that its reuse was critical. Our task-level description language, Durra, therefore evolved from this need for a language to serve as a buffer between the application and the hardware.

Durra is currently being applied to describing a part of an autonomous land vehicle vision application that runs on the existing Carnegie Mellon *Warp machine* [1], which consists of one Sun workstation and a systolic array of ten processors. The Durra compiler generates Unix shell commands that specify which programs to download to which processor at which times, and the location of data and results. A simulator driven by Durra's timing expressions is used for debugging as well as aiding in the design of the next version of a CMU heterogeneous machine.

Our language work is still in an exploratory phase: we consider Durra as a reasonable prototype language that aims to satisfy both application programmers and hardware designers. We encourage use of Durra for other real-time applications developed for other parallel and distributed architectures so that we know best how to trim and change the language.

References

[1] M. Annaratone.
The Architecture of a Systolic Supercomputer.
In *Proceedings of the IEEE-CS Compcon Spring 87 Conference*. IEEE Computer Society Press, February, 1987.

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

[3] M.R. Barbacci and J.M. Wing.
Specifying Functional and Timing Behavior for Real-time Applications.
In *Proceedings of the Conference on Parallel Architectures and Languages Europe -- PARLE*. Springer-Verlag, June, 1987.

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

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

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

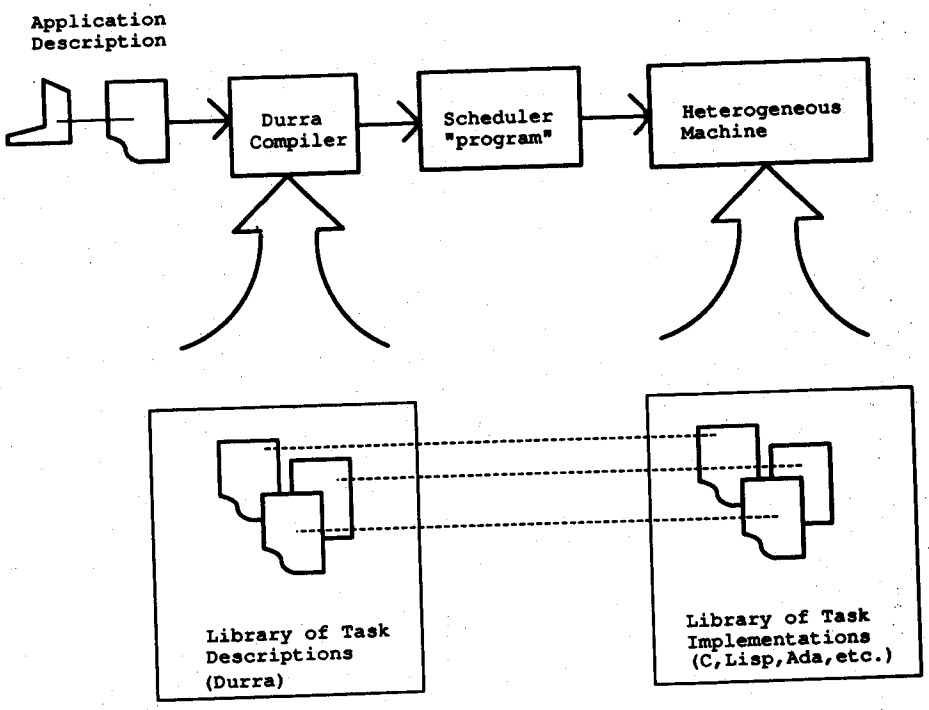

Figure 1 - Scenario for Developing an Application

```
task task-name                          -- Used for communication between a process and a queu
  ports
    port-declarations
  signals                          -- Used for communication between a user process and the schedule
    signal-declarations
  attributes                              -- Used to specefy additional properties of the tas
    attribute-value-pairs
  behavior                         -- A description of the functional and timing behavior of the tas
    requires predicate
    ensures predicate
    timing timing expression
  structure                             -- A graph describing the internal structure of the tas
    process-declarations                     -- Declaration of instances of internal subtask
    bind-declarations                  -- Mapping of internal process ports to this task's port
    queue-declarations                    -- Means of communication between internal processe
    reconfiguration-statements                  -- Dynamic modifications to the structur
end task-name
```

Figure 2 - A Template for Task Descriptions

```
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))
      timing when (~isEmpty(in1) and ~isEmpty(in2)) =>
              ((in1.Dequeue || in2.Dequeue) delay[10,15] out1.Enqueue)
end multiply
```

Figure 3 - The Timing of a Matrix Multiplication Task

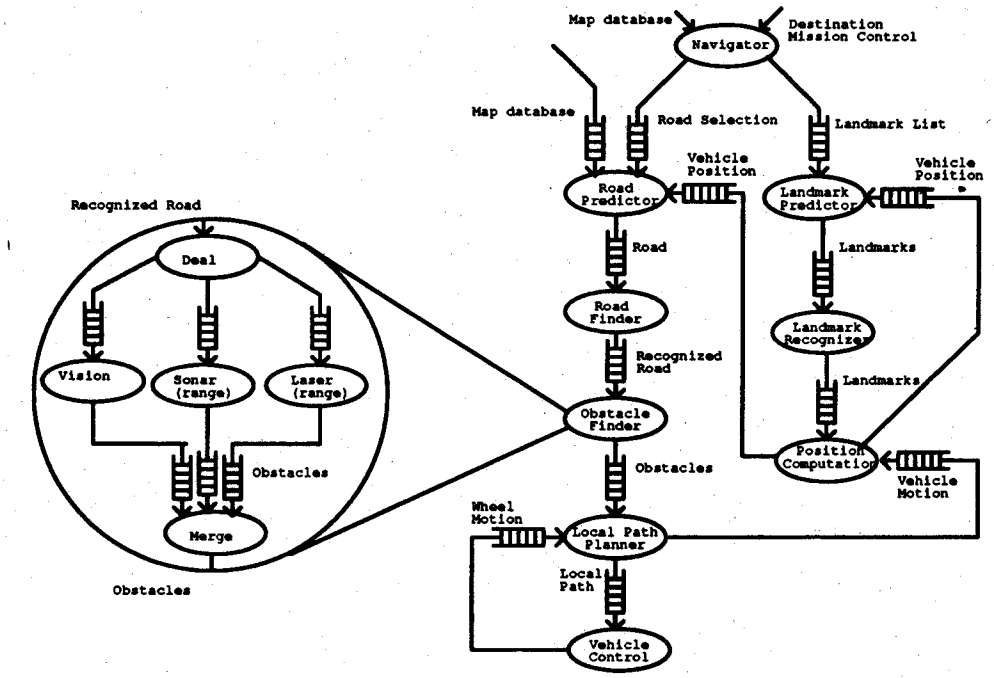

Figure 4 - Process-Queue Graph

```
task obstacle_finder
  ports
    in1: in recognized_road;
    out1: out obstacles;
  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 = warp_1 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;
  reconfiguration
    if Current_Time >= 6:00:00 local and Current_Time < 18:00:00 local then
        process
            p_vision: task vision attributes processor = warp_2; end vision;
        queue
            q5: p_deal.out3   > > p_vision.in1;
            q6: p_vision.out1 > > p_merge.in3;
        end if;
end obstacle_finder;
```

Figure 5 - Structural Information