Durra: Language Support for Large-Grained Parallelism

Mario R. Barbacci\textsuperscript{1,2}, Charles B. Weinstock\textsuperscript{1}, and Jeannette M. Wing\textsuperscript{2}

Carnegie Mellon University
Pittsburgh, PA 15213, U.S.A.

This paper appears in the Proceedings of the International Conference on Parallel Processing and Applications, September 23-25, 1987, L'Aquila, Italy.

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. We present a new language, Durra, to write what we call task-level application descriptions. Durra is used to structure and specify components of a large application program at a high enough level so as to be independent of changing hardware configurations. At the same time, Durra allows application developers to reuse software that has been finely tuned for specific processors. In this paper, we present the scenario for using Durra, its salient linguistic features, and the status of its implementation.

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, to make decisions based on these data, and to send commands to control motors and other physical devices. Since the speed of, and the resources required by 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. Since the patterns of communication can vary over time, and, since 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 and implementation of Durra [2], 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 in some kind of executable "machine language" but rather into commands for a run-time scheduler. We assume therefore that each of the processors in a heterogeneous machine has languages, compilers, libraries of (reusable) programs, and

\textsuperscript{1}Software Engineering Institute \textsuperscript{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 (DARPA) under 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 U.S. Government.
other software development tools that cater to the special properties of a processor's architecture. Durrä's support environment is responsible for coordinating the use and interaction of the separate software environments of the individual processors.

There are three distinct phases in the software development process for a heterogeneous machine: (1) the creation of a library of tasks, (2) the creation of an application description, and finally (3) the execution of the application. During the first phase, the developer breaks the application into specific tasks (e.g., sensor processing, feature recognition, map database management, and route planning) and writes code implementing the tasks. For each implementation of a task, the developer writes a Durrä task description and enters it into the library. Developing programs for some of the more exotic processors involves selecting algorithms appropriate to a processor's architecture, and then painstakingly testing and tuning the code to take advantage of any special features of the processor. For example, an application might use a matrix multiplication task written in assembly for a systolic array processor while simultaneously accessing a database of three-dimensional images maintained by a program written in C running on a workstation. Developing these programs is a slow and difficult process and Durrä facilitates their reuse in multiple applications.

During the second phase, the user writes a Durrä application description. Syntactically, an application description is identical to a compound or structured task description and can be stored in the library and used later as a component task in a larger application description. When the application description is compiled, the compiler generates a set of resource allocation and scheduling commands. During the last phase, the scheduler executes a set of commands which are produced by the compiler. These commands instruct the scheduler to download the task implementations, (i.e., code corresponding to the component tasks) to the processors and issue the appropriate commands to execute the code.

In Section 2, we illustrate the main features of Durrä through examples. In Section 3, we describe the existing implementation of tool support for Durrä. Section 4 includes preliminary conclusions and directions for future work. Further details on the language can be found in the Durrä reference manual [2] and an overview paper [3].

2. Task Descriptions

Task descriptions are the building blocks of the application programs. A task description contains information describing (1) its interface to other tasks (ports) and its attributes, (2) its functional and timing behavior, and (3) its internal structure, thereby allowing for hierarchical task descriptions. In the following three sections we use examples to illustrate these language features.

2.1. Task Interfaces and Attributes

Processes are instances of tasks and communicate through ports. In the example in Figure 1, the task "navigator" has two input ports and two output ports. Each port declaration specifies the direction of data movement and the type of data flowing through the port. Port data types are described independently of the tasks and are also stored in the library. Data types can be scalars (of possible variable length), arrays of types, or unions of types, for example:

```plaintext
type packet is size 128 to 1024;
type tails is array (5 10) of packet;
type mix is union (heads, tails);
```

--- Packets are of variable length.
--- Tails are 5 by 10 arrays of packets.
--- Mix data could be heads or tails.
task navigator

ports
in1: in map_database;  -- An input port receiving data of type map_database.
in2: in destination;
out1: out road_selection; -- An output port sending data of type road_selection.
out2: out landmark_list;

attributes
processor = sun;        -- The processor type required to execute the task.
implementation = "/usr/durra/navigator.o"; -- The location of the object code.

end navigator;

Figure 1: A Simple Task Description

Attributes specify miscellaneous properties of a task. They are a means of indicating pragmas or hints to
the compiler and/or scheduler, and they serve to characterize alternative implementations of a task.
Example attributes include: author, version number, programming language, file name, and processor
type.

2.2. Behavioral Information

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)) =>

end multiply

Figure 2: The Timing of a Matrix Multiplication Task

Behavioral information specifies functional and timing properties of the task. As illustrated in Figure 2, the
functional information 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 consists of a timing expression describing
the behavior of the task in terms of the operations it performs on its input and output ports.

As described in [4], the pre- and post-conditions constitute a simple Larch interface specification [5, 6].
The formal meaning of the combined functional and timing behavior is defined using Jahanian and Mok's
Real-Time Logic [7].

The matrix multiplication task in Figure 2 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 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, and, finally, will deposit some output in the output queue out2.

2.3. Structural Information
The example in Figure 1 describes a simple task. More complex tasks have the structure of a process-queue graph where the internal processes are instances of simpler tasks, thus allowing for hierarchical application descriptions. This is illustrated in Figure 3.

Process declarations take the form process_name: task task_selection, where the task selection consists of the name of a task and, optionally, interface, attributes, and behavior requirements. A task selection is used to select among a number of alternative implementations for the named task. For example, a process declaration like:

```plaintext
navigator: task navigator attributes author = "jmw"; end navigator;
```

specifies a task ("navigator") which might be available in a number of implementations. In this declaration, we select the implementation authored by "jmw." That is, somewhere in the library there must be a task with the same name and an author attribute with value "jmw". The library task description might specify a number of other attributes, but since these are not mentioned in the task selection, they do not intervene in the matching process. In general, a task can be selected from the library by its name alone (if there is only one implementation), 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.

In addition to the user-defined tasks, the language provides a few predefined tasks that can be instantiated as if they were available in the library, although, in reality, these are implemented directly by the compiler and the scheduler. These tasks provide data flow primitives such as deal, broadcast, and merge, which can be instantiated and used like any normal process.

Port binding declarations map ports of the internal processes to ports defining the external interface of a task. In the example in Figure 3b, port in1 of the current task (ALV) is bound (or made equivalent) to port in1 of process road_predictor.

Queue declarations take the form:

```plaintext
name [size]: port_1 > transformation > port_2
```

and specify data flow from output ports (port_1) into input ports (port_2). The size specifies the maximum queue length. The transformations are operations applied to data coming from a source port in order to make them acceptable to a destination port. Arbitrarily complex transformations can be written as separate processes and these processes invoked in the queue declaration. Simple transformations, using predefined operations, can be specified directly in the queue declaration:

```plaintext
q1: p1.out1 > > p2.in1;
q2: p1.out1 > (2 1) transpose > p2.in;
q3[100]: p1.out1 > massager > p2.in1;
```

--- Data flows between processes p1 and p2.
- Data (arrays) are transposed in the queue.
-- Process massager operates on the data in the queue.
task ALV
  ports
  in1, in2: in map_database;
  in3: in destination;
  structure
  process
    navigator:
      local_plan
    road_predictor:
    landmark_predictor:
    . . . . . . .
  ct_process:
  queue
    q1: navigator.out1 > > road_predictor.in2;
    q2: navigator.out2 > > landmark_predictor.in1;
    . . . . . . .
    q12: position_computation.out2 > > landmark_predictor.in2;
  bind
    in1 = road_predictor.in1;
    in2 = navigator.in1;
    in3 = navigator.in2;
end ALV;

b: Durra Text

Figure 3: An Application Description
3. Durra Tools

A prototype implementation of Durra is under development. The effort is divided into two pieces: the compiler and the scheduler. The compiler takes task and type descriptions and produces a set of instructions for executing an application program. The scheduler uses these instructions to direct the execution of the tasks on a heterogeneous machine.

3.1. The Compiler

An application program consists of a set of task and type descriptions. The purpose of the compiler is to turn these descriptions into a set of commands to the scheduler. There are essentially three phases to this operation. In the first phase, individual tasks or type descriptions are parsed into a syntax tree. The second phase uses this syntax tree to do semantic checking and resolve references to external (i.e., library) tasks and types. The third phase generates scheduler commands.

```
--Link V0.1 5/13/1987 17:32:07 task02.durra TASK NAVIGATOR
    (OP_TASKDES s2,2
     NAVIGATOR
    (OP_PORTLIST !3,11
        (OP_IMPORT !3,6
            INIT
            MAP_DATABASE |"--Link V0.1 5/13/1987 17:28:45 type02.durra TYPE MAP_DATABASE"
            (OP_IMPORT !4,6
                IN2
                DESTINATION |"--Link V0.1 5/13/1987 17:28:55 type03.durra TYPE DESTINATION"
                (OP_OUTPORT !5,6
                    OUT1
                    ROAD_SELECTION |"--Link V0.1 5/13/1987 17:29:12 type05.durra TYPE ROAD_SELECT
                    (OP_OUTPORT !6,6
                        OUT2
                        LANDMARK_LIST |"--Link V0.1 5/13/1987 17:29:58 type10.durra TYPE LANDMARK_LIS
                        (OP_ATTRIBUTELIST !8,13
                            (OP_ATTRIBUTE !9,16
                                PROCESSOR
                                SUN)
                            (OP_ATTRIBUTE !11,21
                                IMPLEMENTATION
                                "/usr/durra/navigator.o")
                        ))
        ))
    )
```

Figure 4: The Syntax Tree for Task NAVIGATOR

Figure 4 shows the syntax tree, after linking, of the navigator task of Figure 1. The first line in the figure contains version control information, identifying the version of the compiler, the date of compilation, the input file name, and the task name. The collection of these header lines constitutes the application library directory, which is kept as a separate file and used at compile time to identify library tasks and types.

As the compiler resolves external references, it decorates the syntax tree with a copy of the header lines to facilitate future references. This is shown in Figure 4, where there are four external references, each to a type. In the more general case, there would be references to other tasks. The process is recursive in that the external references themselves must be fully resolved down to the simple type or task level.

Once a Durra application description has all of its external references resolved, the compiler produces instructions that will be used to guide the scheduler's operation. Some of the instructions produced for the ALV application description (Figure 3) are shown in Figure 5. Some instructions tell the scheduler to
allocate ports and queues, as well as what tasks to load on what processors. Other instructions define types so the scheduler can properly allocate memory and tell it what ports are equivalent as a result of a bind declaration.

3.2. The Scheduler

The scheduler is responsible for coordinating the execution of a set of tasks in a heterogeneous machine as specified by instructions output by the Durra compiler. These instructions are used to initialize the heterogeneous machine: task_load instructions tell the scheduler what tasks are to run on what machines, buffer_task instructions tell the scheduler to instantiate pre-defined tasks (e.g. the deal and merge tasks). port_allocate and queueAllocate instructions, tell the scheduler how to allocate space for data, type instructions describe data. Finally, transformation instructions tell the scheduler what data transformations will be associated with each queue, if any.

We do not expect that the languages that will be used for programming a heterogeneous machine will have constructs that map directly onto the Durra concept of a port. Instead, we provide for each language a procedural interface consisting of four procedures that a task can call for sending data to and receiving data from ports. These procedures are described in Table 3-1.

<table>
<thead>
<tr>
<th>Procedure Call</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>PortID(Name)</td>
<td>Takes a port Name and returns a unique descriptor (ID) to be used for further references to that port.</td>
</tr>
<tr>
<td>Put_Port(ID,Data,Count)</td>
<td>Sends Count bits at address Data to port ID.</td>
</tr>
<tr>
<td>Get_Port(ID,Data,Count)</td>
<td>Gets Count bits at address Data from port ID.</td>
</tr>
<tr>
<td>Test_Port(ID)</td>
<td>Determines if data is available on port ID.</td>
</tr>
</tbody>
</table>

Table 3-1: Procedures to Operate on Data Ports

These four procedures are the initial set that will be provided for any programming language used to implement Durra tasks. Tasks running on the heterogeneous machine processors communicate with the scheduler using a Remote Procedure Call protocol built on top of the TCP/IP communications protocol [8].
When all the tasks are loaded, and the ports and queues are allocated, the scheduler directs them to begin execution. A task that needs input will wait for output from the task that produces it. Others will produce output, which the scheduler will pass on to waiting tasks.

4. Conclusions and Future Work
Our original motivation for designing and implementing a task-level language was to fill a need in two communities:

- That of 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

- That of 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) become 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 evolved from this need for a language to serve as a buffer between the application and the hardware.

We are currently adding the following features to the current Durra environment:

1. Runtime support for dynamic reconfiguration -- The language provides reconfiguration statements which are directives to the scheduler. They specify changes to the structure of a task and the conditions under which these changes take effect. Typically, a number of existing processes and queues are removed from the graph, and a number of new processes and queues are declared and connected to the remainder of the original graph. The reconfiguration condition is a boolean expression involving time values, queue sizes, and other information available to the scheduler at run time.

2. A collection of built-in data transformations such as matrix transpose and corner turning.

3. A procedural interface (as described in Section 3.2) to the scheduler for each of the individual programming languages, e.g., C, Common Lisp, and assembly, that run on the separate processors;

Durra is currently being used to describe a part of an autonomous land vehicle vision application that runs on a configuration of Warp machines (a systolic array of ten processors with one Sun workstation as host [1]) and Sun workstations connected via an Ethernet.

We consider Durra to be a reasonable prototype language that aims to satisfy both application programmers and hardware designers, and we encourage other researchers to use Durra for their specific applications and architectures and to provide us with feedback on their experience.
References

The Architecture of a Systolic Supercomputer.

*Durra: A Task-Level Description Language.*

*Durra: A Task-level Description Language.*


*Larch in Five Easy Pieces.*

The Larch Family of Specification Languages.


*Military Standards For DoD Internet Protocols.*
Naval Publications and Forms Center. Code 3051, 5801 Tabor Avenue, Philadelphia, PA 19120,