Developing Applications for Heterogeneous Machine Networks: The Durra Environment

Mario R. Barbacci, Dennis L. Doubleday,
Charles B. Weinstock, and
Jeannette M. Wing
Carnegie Mellon University

ABSTRACT: In this paper we describe Durra, a language designed to support PMS-level programming, and its runtime environment.

Users of networks of heterogeneous processors are concerned with allocating specialized resources to tasks of medium to large size. They need to create processes, which are instances of tasks, allocate these processes to processors, and specify the communication patterns between processes. These activities constitute *Processor-Memory-Switch (PMS) Level Programming*, in contrast with traditional programming activities, which take place at the *Instruction Set Processor (ISP) Level*.

This work is sponsored by the U.S. Department of Defense. The views and conclusions contained in this document are solely those of the author(s) and should not be interpreted as representing official policies, either expressed or implied, of Carnegie Mellon University, the U.S. Air Force, the Department of Defense, or the U.S. Government.

An application or PMS-level program is written in Durra as a set of *task descriptions* and *type declarations* that prescribes a way to manage the resources of a heterogeneous machine network. The application describes the tasks to be instantiated and executed as concurrent processes, the types of data to be exchanged by the processes, and the intermediate queues required to store the data as they move from producer to consumer processes.

The environment consists of three active components: the application tasks, the Durra server, and the Durra scheduler. After compiling the type declarations, the component task descriptions, and the application description, the application can be executed by starting an instance of the server on each processor, starting an instance of the scheduler on one of the processors, and downloading the component *task implementations* (i.e., the programs) to the processors. The scheduler receives as an argument the name of the file containing the scheduler program generated by the compilation of the application description. This step initiates the execution of the application.

---

1. *Programming Heterogeneous Machines*

It is becoming commonplace to have a computing environment consisting of loosely-connected networks of multiple special- and general-purpose processors. We call such an environment a *heterogeneous machine*. These machines are of special interest to developers of real-time, embedded applications in which many concurrent, large-grained tasks or programs cooperate to process data obtained from physical sensors, make decisions based on these data, and send commands to control motors and other

physical devices. During execution time, these tasks are instantiated as concurrent processes, running on possibly separate processors and communicating with each other by sending messages of different types.

Since the patterns of communication between these processes can vary over time and the speeds of the individual processors can differ widely, developers of applications running on a heterogeneous machine need to control the allocation of processors to processes in order to meet throughput requirements. Processors are not the only critical resource. In addition to special purpose processors such as systolic arrays, and general-purpose workstations, the resources that must be allocated include fast switches, data buffers with processing capabilities, etc., as illustrated in Figure 1. Currently, users of a heterogeneous machine follow the

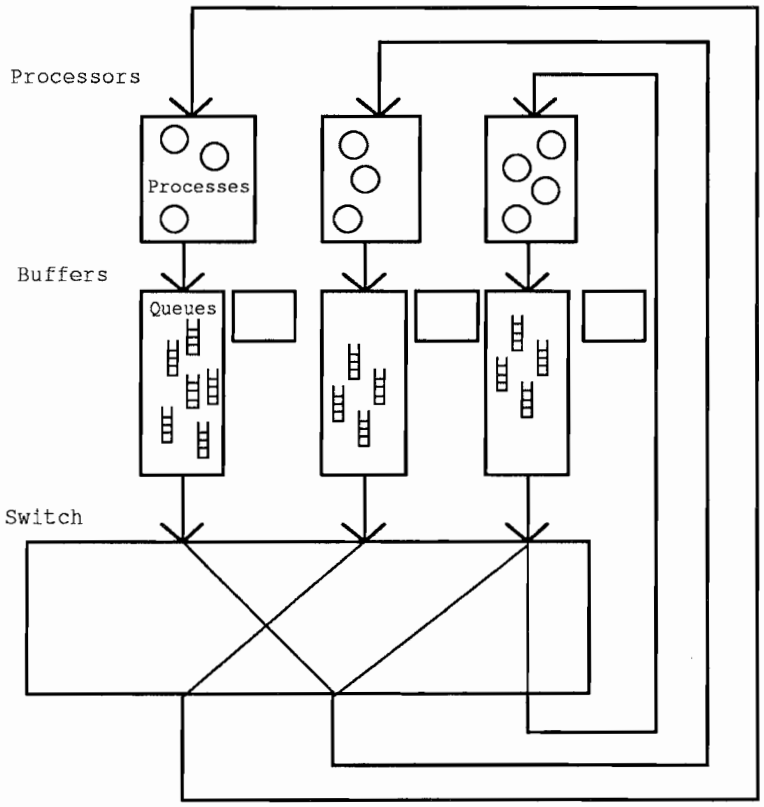

Figure 1: A Heterogeneous Machine

same pattern of program development as users of conventional processors: users write individual tasks as separate programs, in the different programming languages (e.g., C, Lisp, Pascal) supported by the processors, and then hand code the allocation of resources to their application by explicitly loading specific programs to run on specific processors at specific times.

We claim that developing software for a heterogeneous machine is qualitatively different from developing software for conventional processors. It requires different kinds of languages, tools, and methodologies; and in this paper we address some of these issues by presenting a language, Durra, and its support tools (compiler, runtime environment, and task emulator).

2. *Introduction to Durra*

Durra [Barbacci & Wing 1986; Barbacci et al. 1988a] is a language designed to support PMS-level programming. PMS stands for Processor-Memory-Switch, the name of the highest level in the hierarchy of digital systems introduced by Bell and Newell [1971]. An application or PMS-level program is written in Durra as a set of *task descriptions* and *type declarations* that prescribes a way to manage the resources of a heterogeneous machine network. The application describes the tasks to be instantiated and executed as concurrent processes, the types of data to be exchanged by the processes, and the intermediate queues required to store the data as they move from producer to consumer processes. Because tasks are the primary building blocks, we refer to Durra as a *task-level description language*. We use the term "description language" rather than "programming language" to emphasize that a Durra application is not translated into object code in some kind of executable (conventional) "machine language" (the domain of the Instruction Set Processor or ISP level introduced in Bell and Newell [1971]). Instead, a Durra application is a description of the structure and behavior of a logical machine to be synthesized into resource allocation and scheduling directives, which are then interpreted by a combination of software, firmware, and hardware in each of the processors and buffers of a

heterogeneous machine (the domain of PMS). This is the translation process depicted in Figure 2a.

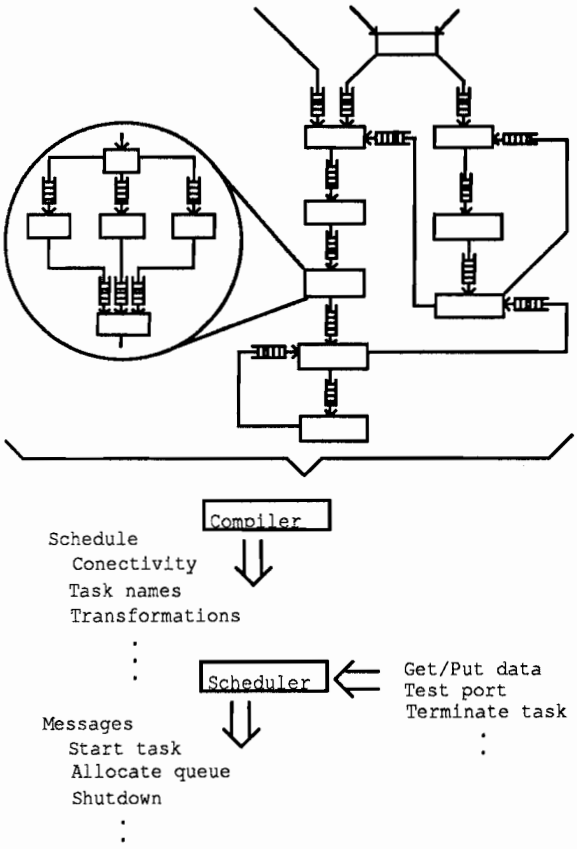

Figure 2a: Compilation of a PMS-Level Program Graph

2.1 Scenario for Developing an Application

We see three distinct phases in the process of developing an application using Durra: the creation of a library of tasks, the creation of an application using library tasks, and the execution of the application. These three phases are illustrated in Figure 2b.

During the first phase, the developer of the application writes descriptions of the data types (image buffers, map database queries, etc.) and of the tasks (sensor processing, feature recognition, map database management, etc.).

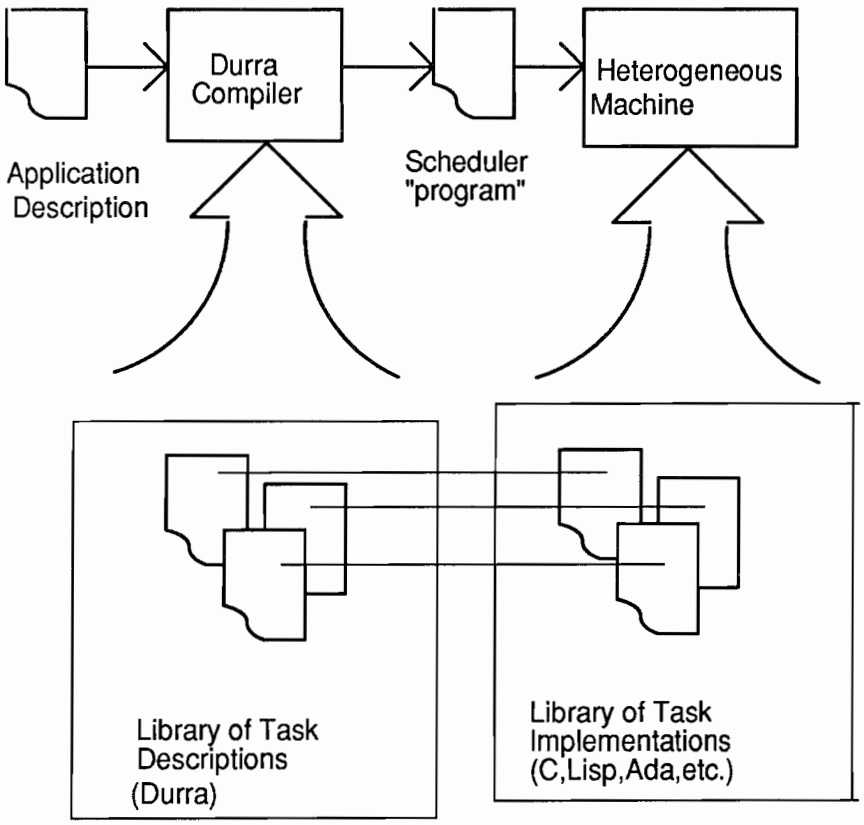

Figure 2b: Developing a Durra Application

Type declarations are used to specify the format and properties of the data that will be produced and consumed by the tasks in the application. As we will see later in this section, tasks communicate through typed ports; and for each data type in the application, a type declaration must be written in Durra, compiled, and entered in the library.

Task descriptions are used to specify the properties of a task implementation (a program). For a given task, there may be many implementations, differing in programming language (e.g., C or assembly language), processor type (e.g., Motorola 68020 or DEC VAX), performance characteristics, or other attributes. As in the case of type declaration, for each implementation of a task, a task description must be written in Durra, compiled, and entered in the library. A task description includes specifications of a task

implementation's performance and functionality, the types of data it produces or consumes, the ports it uses to communicate with other tasks, and other miscellaneous attributes of the implementation.

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

During the last phase, the scheduler loads the task implementations (i.e., programs corresponding to the component tasks) into the processors and issues the appropriate commands to execute the programs. The operation of the scheduler and the other runtime environment components are described in the following section.

2.2 Task Descriptions

Task descriptions are the building blocks for applications. Task descriptions include the following information (Figure 3): (1) its interface to other tasks (*ports*); (2) its *attributes*; (3) its functional and timing *behavior*; and (4) its internal *structure*, thereby allowing for hierarchical task descriptions.

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

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

  behavior -- Used to specify functional and
                -- timing behavior of the task
    requires predicate
    ensures predicate
    timing timing expression
  structure -- A graph describing the
                -- internal structure of the task
    process-declarations  -- Declaration of
                -- instances of internal subtasks
```

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

    queue-declarations -- Means of
       -- communication between internal processes

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

Figure 3: A Template for Task Descriptions

Interface information – This portion defines the ports of the processes instantiated from the task.

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

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

Attribute Information – This portion specifies miscellaneous properties of a task. Attributes 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 actual value of a property; in a task selection, the user of a task lists the desired value of the property. Example attributes include author, version number, programming language, file name, and processor type:

```
attributes
   author = "jmw";
   implementation = "program_name";
   Queue_Size = 25;
```

Behavioral Information – This portion specifies functional and timing properties about the task. The functional information part of a task description consists of a pre-condition on what is required to be true of the data coming through the input ports, and a post-condition on what is guaranteed to be true of the data going out through the output ports. The timing expression describes the behavior of the task in terms of the operations it performs on its input and output ports. For additional information about the syntax and semantics of the functional and timing behavior description, see the Durra reference manual [Barbacci & Wing 1986].

Structural Information – This portion defines a process-queue graph (e.g., Figure 2a) 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 4, which shows the Durra (i.e., textual) version of the example in Figure 2a.

```
task ALV
  ports
    in1, in2: in map_database;
    in3: in destination;
  structure
    process
      navigator:  task navigator
                  attributes author = "jmw";
                  end navigator;
      road_predictor:   task road_predictor;
      landmark_predictor: task
                          landmark_predictor;
      . . . . . .
      ct_process:         task corner_turning;
    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;
```

Figure 4: Structural Information

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, 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 hierarchically structured tasks.

A queue declaration of the form

queue_name [*queue_size*]: *port_name_1* > *data_transformation* > *port_name_2*

creates a queue through which data flow from an output port of a process (port_name_1) into the input port of another process (port_name_2). Data transformations are operations applied to data coming from a source port before they are delivered to a destination port.

A port binding of the form

task_port = *process_port*

maps a port on an internal process to a port defining the external interface of a compound 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 of the application (i.e., process-queue graph) 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 runtime.

3. The Durra Runtime Environment

There are three active components in the Durra runtime environment: the application tasks, the Durra server, and the Durra scheduler. Figure 5 shows the relationship among these components.

After compiling the type declarations, the component task descriptions, and the application description, as described

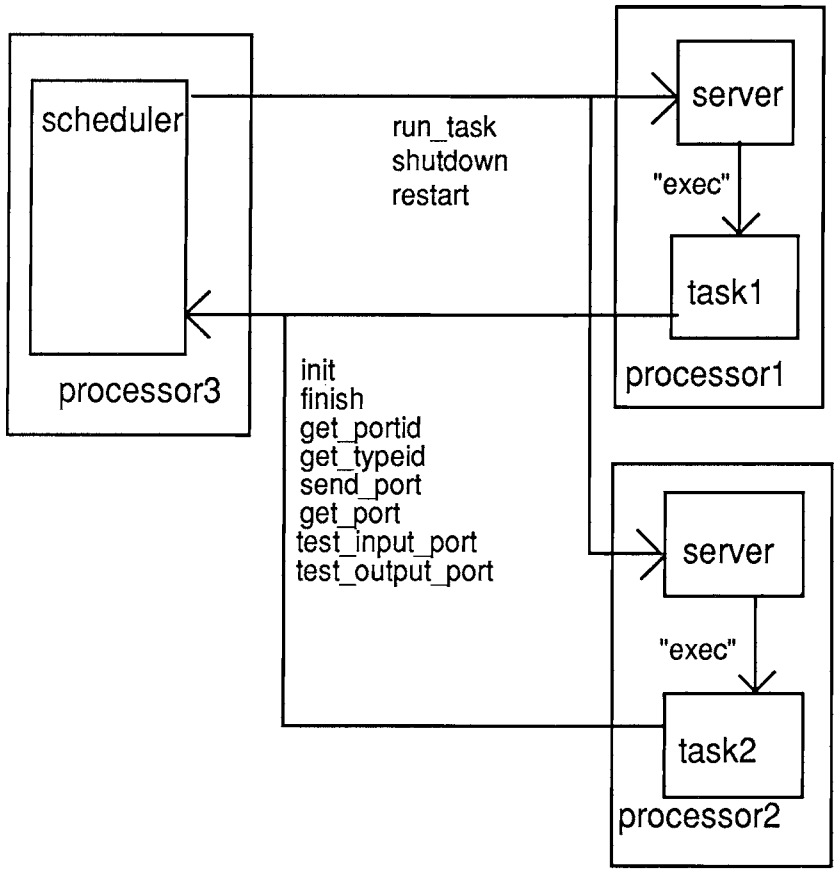

Figure 5: The Durra Runtime Environment

previously and illustrated in Figure 2, the application can be executed by performing the following operations:

1. The component task implementations (section 3.3) must be stored in the appropriate processors, in some directory known to the Durra servers and scheduler.
2. An instance of the Durra server (section 3.2) must be started in each processor.
3. The scheduler (section 3.1) must be started in one of the processors. The scheduler receives as an argument the name of the file containing the scheduler program generated by the compilation of the application description. This step initiates the execution of the application.

In the remainder of this section, we sketch the three components of the runtime environment: the scheduler, the server, and the application task. For additional details, see [Barbacci et al. 1988b].

3.1 The Scheduler

The scheduler is the part of the Durra runtime system responsible for starting the tasks, establishing communication links, and monitoring the execution of the application. In addition, the scheduler implements the predefined tasks (*broadcast*, *merge*, and *deal*) and the data transformations described in [Barbacci & Wing 1986]. The scheduler is invoked with the name of the file containing the scheduler instructions generated by the Durra compiler. These instructions describe the programs to be executed as concurrent processes, the ports and queues used for communication between these processes, the data types exchanged between these tasks, and the possible reconfigurations to the structure of the application. These recionfigurations consist of elimination of existing processes, activitation of new processes, and conection of these new processes to the remaining network.

In the current implementation, the scheduler runs under UNIX and the servers and task applications run under UNIX or VMS but are easily portable to any operating system suporting the TCP/IP protocol. The scheduler takes advantage of UNIX communication primitives to allocate sockets for receiving remote procedures calls from the application tasks, as described in section 3.3.

3.2 The Server

The server is responsible for starting tasks on its corresponding processor, as directed by the scheduler. One instance of the server must be running on each processor that is to (potentially) execute Durra tasks.

When a server begins execution, it listens in on a predetermined socket for messages from the scheduler. Once a communication channel is open, the scheduler can send to the server requests to start and kill user tasks.

3.3 Application Tasks

The component task implementations making up a Durra application can be written in any language for which a Durra interface has been provided. As of this writing, there are Durra interfaces for both C and Ada (see [Barbacci et al. 1988b] for details).

When a task is started, the scheduler supplies it, via the server, with the following information: the name of the host on which the scheduler is executing, the UNIX socket on which the scheduler is listening for communications from the task, and a small integer to be used in identifying the task. These parameters are necessary to establish proper communication with the scheduler.

Application tasks use the interface to communicate with other tasks. From the point of view of the task implementation, this communication is accomplished via procedure calls, which return only when the operation is completed. The interface provides remote procedure calls (RPCs) to initialize and terminate communications with the scheduler, to request port identifiers, to send and receive data on specific ports, and to test the contents of the queues attached to the task ports.

Using this collection of scheduler calls, application tasks typically would exhibit the following behavior:

1. Establish communication with the scheduler.
2. Request port identifiers. These are tokens or capabilities that uniquely identify the ports.
3. Send and receive data.
4. Break communication with the scheduler.

This behavior is illustrated by an example application in the next section.

4. A Durra Example

This section contains a complete example of a Durra application. It consists of two type declarations, three component task

descriptions (and their implementations), the application description, and the scheduler instructions produced by the Durra compiler.

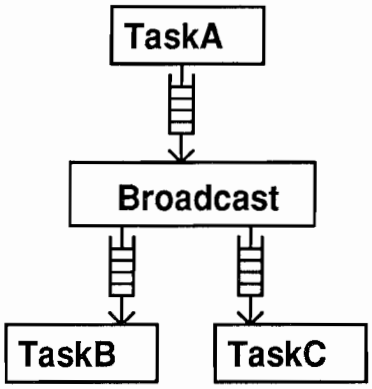

Figure 6: Application Structure

The example (Figures 6 and 7) illustrates the use of a predefined task, *broadcast*, which is implemented directly by the scheduler. In this application, one task ("taska") is sending out strings of data, and the *broadcast* buffer task is sending it on to two other tasks ("taskb" and "taskc"). The application description (cements all of the component tasks together.

```
type byte is size 8;

type string is array of byte;
```

a – Type Declarations

```
task taska
ports
    out1: out string;
attributes
    processor = vax;
    implementation = "source_task";
end taska;

task taskb
ports
    in1: in string;
attributes
    processor = vax;
    implementation = "sink_task";
end taskb;
```

```
task taskc
ports
    in1: in string;
attributes
    processor = vax;
    implementation = "sink_task";
end taskc;
```

b – Component Task Descriptions

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

c – Application Description

Figure 7: Durra Type Declarations and Task Descriptions

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

"Taska" has a single output port, "out1," which produces strings. It can run on any VAX processor and is implemented by the program "source_task." "Taskb" and "taskc" both have a single input port, "in1," which consume strings. These two tasks are implemented by the program "sink_task."

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

After all of the above files are compiled, the Durra compiler generates a file with instructions to the scheduler. See section 3.1 for further information about the scheduler instructions. For this example application, the compiler produces the scheduler instructions shown in Figure 8.

```
(buffer_task TOP MAIN.PB BROADCAST)
(port_allocate TOP MAIN.P1 OUT1 STRING out)
(port_allocate TOP MAIN.P2 IN1 STRING in)
(port_allocate TOP MAIN.P3 IN1 STRING in)
(port_allocate TOP MAIN.PB IN1 STRING in)
(port_allocate TOP MAIN.PB OUT1 STRING out)
(port_allocate TOP MAIN.PB OUT2 STRING out)
(queue_allocate TOP MAIN Q1B P1 OUT1 PB IN1 0)
(queue_allocate TOP MAIN QB2 PB OUT1 P2 IN1 0)
(queue_allocate TOP MAIN QB3 PB OUT2 P3 IN1 0)
(task_attribute TOP MAIN SOURCE "taskmain.durra.TREE")
(task_attribute TOP MAIN.P1 IMPLEMENTATION "source_task")
(task_attribute TOP MAIN.P1 PROCESSOR "VAX")
(task_attribute TOP MAIN.P1 SOURCE "taska.durra.TREE")
(task_attribute TOP MAIN.P2 IMPLEMENTATION "sink_task")
(task_attribute TOP MAIN.P2 PROCESSOR "VAX")
(task_attribute TOP MAIN.P2 SOURCE "taskb.durra.TREE")
(task_attribute TOP MAIN.P3 IMPLEMENTATION "sink_task")
(task_attribute TOP MAIN.P3 PROCESSOR "VAX")
(task_attribute TOP MAIN.P3 SOURCE "taskc.durra.TREE")
(type BYTE SIZE 8 8)
(type STRING ARRAY BYTE)
```

Figure 8: Scheduler Instructions

The *buffer_task* instruction indicates which buffer task to use as process "pb." Buffer tasks are those tasks predefined in the language (*broadcast*, *merge*, and *deal*).

The *port_allocate* instructions set up all the ports in the application. Recall that port names are relative to a process and therefore do not have to be unique across the application.

The *queue_allocate* instructions set up all the queues in the application. For instance one of the instructions allocates the queue named "Q1B," taking input from port "OUT2" of process "MAIN_P1," and outputting to port "IN1" of process "MAIN_PB." The queue has the default queue size.

The *task_attribute* instructions contain the various attributes (name/value pairs) specified in the task descriptions.

The *task_load* instructions associate the program to run (the task implementation), the processor class, and the process name.

The *type* instructions define the application data types "BYTE" and "STRING."

Finally, we complete the example by listing the task implementations for "source_task" and "sink_task." These are written in Ada and are shown in Figures 9 and 10. The interesting

statements are the calls to the procedures defined in package "interface." This package implements the remote procedure calls that allow user tasks written in Ada to communicate with the Durra scheduler.

```
with System;
with Interface;
procedure source_task is
  ------------------------------------------------
  --| A source task has one output port, "out1".
  --| Its behavior is to loop
  --| sending 100 strings to out1.
  ------------------------------------------------
  max_message_size: constant integer := 1000;
  message_buffer: string(1..max_message_size) :=
      (others => ' ');
  out1_port_id, out1_bound: positive;
  out1_type_id, out1_type_size: natural;
begin
  Interface.Init;
  Interface.Get_PortID("out1", out1_port_id,
      out1_bound, out1_type_size);
  Interface.Get_TypeID("string", out1_type_id,
      out1_type_size);
  for i in 1..100
  loop
    interface.Send_Port(out1_port_id,
                  message_buffer(1)'address,
                  max_message_size, --| the real thing
                  out1_type_id);
  end loop;
  Interface.Finish;
end source_task;
```

Figure 9: Ada Task Implementation of source_task

```
with System;
with Interface;
procedure sink_task is
  ------------------------------------------------
  --| A sink task has one input port, "in1".
  --| Its behavior is to loop
  --| receiving 100 strings from in1.
  ------------------------------------------------
  max_message_size: constant integer := 10000;
  message_buffer: string(1..max_message_size);
  in1_port_id, in1_bound: positive;
  in1_type_id, in1_type_size: natural;
  actual_message_type_id, actual_message_size: natural;
```

```
begin
  Interface.Init;
  Interface.Get_PortID("in1", in1_port_id, in1_bound,
      in1_type_size);
  Interface.Get_TypeID("string", in1_type_id,
      in1_type_size);
  for i in 1..100
  loop
    Interface.Get_Port(in1_port_id,
                  message_buffer(1)'address,
                  actual_message_size,
                  actual_message_type_id);
  end loop;
  Interface.Finish;
end sink_task;
```

Figure 10: Ada Task Implementation of sink_task

5. *Debugging Tools*

Testing and debugging programs running on a heterogeneous machine present many of the same problems that are found with any collection of cooperating processes running asynchronously. In our initial implementation, the problems are alleviated somewhat by the (logically) central scheduler, which controls the passage of information between processes. Nevertheless, special-purpose tools must be provided to facilitate testing and debugging.

The primary debugging facility provided by the Durra runtime environment is the scheduler itself. It provides input and output ports, just like the normal processes, but these ports are used to communicate with the user. Specifically, through the use of these ports, the user can do the following:

- Watch the flow of data through queues.
- Observe the status of each process.
- Inspect and manipulate data coming into or going out of specific ports.
- Force reconfiguration (when reconfiguration is implemented)

Of course, Durra tasks will actually do the communication; and we expect to develop more elaborate debugging facilities using the scheduler's ports and a flexible window manager, allowing the user to watch several processes at once.

In addition to the debugging facilities built into the scheduler, a program that acts as a "universal" task emulator is available for building prototypes of applications. This program, MasterTask [Barbacci 1988], can emulate any task in an application by interpreting the timing expression describing the behavior of the task, performing the input and output port operations in the proper sequence and at the proper time.

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

A Durra timing expression can contain concurrent events as well as loops and guards that block execution until some condition is met (e.g., some amount of time has elapsed since the start of the application, an input queue has a given number of data elements). When MasterTask starts, it reads the Durra timing expression syntax tree for the task it wants to emulate and assigns a light-weight process (an Ada task object in the current implementation) to each node of the tree. This process is responsible for performing one or more node-dependent operations: 1) execute a queue operation (including 2) evaluate a guard expression (including 3) direct the execution of the processes responsible for the subtrees rooted at this node.

Generally, MasterTask exhibits the same behavior as a regular Durra task implementation, issuing the same type of remote procedure calls to the scheduler (see [Barbacci et al. 1988b] for a description of the operations).

6. *Related Work*

Two other languages/systems, DICON [Lee & Goldwasser 1985] and CONIC [Magee & Kramer 1983], can be considered as PMS-level programming languages similar to Durra. Lee and Goldwasser's DICON is a configuration language used to glue together a set of sequential programs written in Prolog or C to

form a distributed program. DICON allows a close coupling of the processes, including passing of pointers to structured data (lists, trees, etc.), which are then used by the interprocess communication servers to retrieve and copy the data. Programs are not as independent from each other as they are in Durra (e.g., according to [Lee & Goldwasser 1985], C programs need to know if they are communicating with a Prolog program and are restricted in the types of data they can send or receive). A DICON configuration specification includes process specifications but apparently without the full flexibility of Durra to use various types of function, timing, or attribute information to characterize and retrieve library tasks. The DICON compiler attempts to find a nearly optimal process allocation and scheduling strategy for a given configuration specification. This is in contrast to Durra, in which the allocation and scheduling strategies are under control of the application developer, including the possible dynamic reconfiguration of the logical network.

Magee and Kramer address the problem of dynamic reconfiguration of real-time systems in the design of the CONIC language. CONIC restricts tasks to be programmed in a fixed language (an extension to Pascal with message passing primitives) running on homogeneous workstations.

Belzile et al. [1986] introduce RNET, a facility for building distributed real-time programs. An RNET program consists of a configuration specification and the procedural code, which is compiled, linked with a run-time kernel, and loaded onto the target system for execution. The language provides facilities for specifying real-time properties, such as deadlines and delays that are used for monitoring and scheduling the processes. These features place RNET at a lower level of abstraction, and thus RNET cannot be compared directly to Durra. Rather, it can be considered as a suitable language for developing the schedulers required by Durra and other languages in which the concurrent tasks are treated as black boxes.

Mamrak et al. [1982] address the need for transforming data exchanged between heterogeneous processors. They describe a system based on canonical representations of data used as an exchange format and the desirable properties of such canonical representations. Durra is not concerned with specific data

formats; rather, it provides a mechanism for invoking arbitrary data transformations as needed. In other words, Durra operates above the level of the canonical representation, if any, and assumes only that data comes in blocks of variable length. Code to transform these blocks has to be provided by the users. However, Mamrak et al. describe a technique for providing a uniform frontend to tools in a distributed environment. This and other similar facilities could be adopted by the application developers without difficulty as Durra operates at a higher level of abstraction.

7. *A Note on Software Development Methodologies*

A great deal of effort has been devoted to the development of improved software development process models. As described in Boehm [1988], models have evolved from the early "code-and-fix" model, through the "stagewise" and "waterfall" models (which attempt to bring order to the process by recognizing formal steps in the process), through the "evolutionary" and "transform" models (which attempt to address the need for experimentation, refinement of requirements, and automation of the code generation phase.) The spiral model of the software process has evolved over several years at TRW, based on experience on a number of large software projects and, as indicated by Boehm, accommodates most previous models as special cases.

The spiral model is basically a refinement of the classical waterfall model, providing for successive applications of the original model (requirements, design, development, testing, etc.) to progressively more concrete versions of the final product.

One of the advantages of this model is that it allows the identification of areas of uncertainty that are significant sources of risk. Once these critical areas are identified, the spiral model allows for the selective application of an appropriate development strategy to these risk areas first. Thus, while at first sight the spiral model looks like no better than the waterfall model, a key difference is that the spiral allows the designers to concentrate in selected problem areas rather than following a predetermined

order. Once the higher-risk problem has been taken care off, the next higher-risk area can be attacked, and so on.

To be successful, any approach based on successive refinements, such as the spiral model, must be supported by tools appropriate to the task on hand. Users of the spiral model must be able to selectively identify high-risk components of the product, establish their requirements, and then carry out the design, coding, and testing phases. Notice that it is not necessary that this process be carried out through the testing phase – higher-risk components might be identified in the process and these components must be given higher priority, suspending the development process of the formerly riskier component.

The PMS-level programming paradigm we have described in this paper fits very naturally this style of software development. Although we don't claim to have solved all problems or identified all the necessary tools, we would like to suggest that a language like Durra would be of great value in the context of the spiral model. It would allow the designer to build mock-ups of an application, starting with a gross decomposition into tasks described by templates specified by their interface and behavioral properties. Once this is completed, the application can be emulated using MasterTask as a stand-in for the yet-to-be written task implementations.

The result of the emulation would identify areas of risk in the form of tasks whose timing expressions suggest are more critical or demanding. In other words, the purpose of this initial emulation is to identify the component task more likely to affect the performance of the entire system. The designers can then experiment by writing alternative behavioral specifications for the offending task until a satisfactory specification (i.e., template) is obtained. Once this is achieved, the designers can proceed by replacing the original task descriptions with more detailed templates, consisting of internal tasks and queues, using the structure description features of Durra. These more refined application descriptions can again be emulated, experimenting with alternative behavioral specifications of the internal tasks, until a satisfactory internal structure (i.e., decomposition) has been achieved. This process can be repeated as often as necessary, varying the degree of refinement of the tasks, and even backtracking if a

dead-end is reached. It is not necessary to start coding a task until later, when its specifications are acceptable, and when it is decided that it should not be further decomposed.

Of course, it is quite possible that a satisfactory specification might be impossible to meet and a task implementation might have to rejected. The designers would then have to backtrack to an earlier, less detailed design and try alternative specifications, or even alternative decompositions of a parent subsystem. This is possible because we are following a strictly top-down approach. The effect of a change in an inner task would be reflected in its impact on the behavioral specifications of a "parent" task. The damage is, in sense, contained and can not spread to other parts of the design.

8. Conclusions

PMS-level programming, as implemented by Durra, lifts the level of programming at the code level to programming at the specification level. What then constitutes a *specification* (e.g., Durra task description) and its *satisfaction* (e.g., Durra task selection) determines the power of programming at the specification level. If a specification is just a list of filenames and their version numbers, then a "program" is simple, and programming is not very powerful: selection of programs from a library indexed by filename is trivial. If a specification includes semantic information, e.g., functional behavior of a task, then programming is quite complex: selection of programs may involve theorem-proving capability. We designed Durra with the ultimate goal of exploiting the rich semantic information included in a task description. For our prototype implementation, however, we have sacrificed semantic complexity in favor of simpler task selection based on interface and attribute information. We gain the advantage of being able immediately to instantiate our general idea of PMS-level programming with a real environment (Durra compiler, runtime system, and task emulator) that runs on a heterogeneous machine (various kinds of workstations connected via an Ethernet). Hence instead of a paper design, we can claim the existence of a working system.

Appendix A: The C Interface Specification

```
/* Header file defining Durra Interface operations */

/* Define short external names for linker */

#define get_portid      getpid
#define get_typeid      gettid
#define get_port        getprt
#define send_port       sndprt
#define test_input_port tstinprt
#define test_output_port tstoutprt

extern void init();
/* Synopsis: (Open a communications channel between the
             Durra Scheduler and an application task.)
void init(initname,     -- name of the application task
          inithost,     -- host where scheduler is running
          initport,     -- address of the socket where the
                        -- Scheduler will be listening
                        -- for traffic
          initidentity, -- ID by which task is known
                        -- to Scheduler
          initdebug);   -- Debugging level
  char *initname;
  char *inithost;
  char *initport;
  char *initidentity;
  char *initdebug;
*/

extern void finish();
/* Synopsis: (Inform the Scheduler that an application
             task is terminating.)

    -- No Parameters
*/

extern void get_portid();
/* Synopsis: (Return the unique ID, the associated
             queue size bound, and the data size
             expected for the specified port.)
void get_portid(name,
                id,
                bound,
                size)
  char *name;
  int *id, *bound, *size;
*/
```

```
extern void get_typeid();
/* Synopsis: (Return the Scheduler-assigned ID and size
             (in bytes) of the specified data type.)
void get_typeid(name,
                typeid,
                size)
  char *name;
  int *typeid, *size;
*/

extern void send_port();
/* Synopsis: (Send "count" bytes of data to the port
             with the specified ID.
             If "ty" is nonzero, then the data is of
             the type with typeid "ty",
             otherwise the type is unspecified.)
void send_port(id,
               data,
               count,
               ty)
  int id,count,ty;
  char *data;
*/

extern void get_port();
/* Synopsis: (Get "count" bytes of data (or the actual
             number of bytes sent, if it is less)
             from the port with the specified ID.
             If "ty" is nonzero, then the data is of
             the type with typeid "ty", otherwise the
             type is unspecified.)
void get_port(id,
              data,
              count,
              ty)
  int id;
  int *ty,*count;
  char *data;
*/

extern int test_input_port();
/* Synopsis: (Return zero if a get_port from the
             specified port will block; otherwise
             return the positive number indicating
             the number of free spaces in the
             associated queue.  Also return the type
             of the next element in next_type,
             and it's size in next_size)
int test_input_port(id,next_type,next_size)
  int id,*next_type,*next_size;
*/
```

```
extern int test_output_port();
/* Synopsis: (Return zero if a send_port to the
             specified port will block; otherwise
             return the positive number indicating
             the number of free spaces in the
             associated queue.)
int test_output_port(id)
  int id;
*/
```

Appendix B: The Ada Interface Specification

```
with system; use system;
-------------------------------------------------
package Interface is
 -------------------------------------------------
 --| Durra Scheduler Interface (Low Level)
 --|
 --| This package provides the interface to the
 --| Durra scheduler for tasks written in Ada.
 --|
 --| The init_* variables are the parameters passed
 --| by the server when a task is initialized.
 --| The server in turn gets them from the scheduler.
 -------------------------------------------------

 -- REVISION HISTORY
 -- 01/03/88   mrb    Package spec created.
 -- 06/13/88   dd     Test_Port expanded to separate
 --                     routines for
 --                     input and output ports.
 -- 07/5/88    dd     Constant environment names
 --                     changed to function calls.

 function User_Task_Name        return STRING;
 function Scheduler_Host        return STRING;
 function Scheduler_Port        return STRING;
 function User_Process_ID       return STRING;
 function Scheduler_Debug_Level return STRING;
 function User_Source_Parameter return STRING;

 procedure Finish;

 procedure Get_Port (Port_ID       : in     POSITIVE;
                     Data          : in System.Address;
                     Data_Size     :    out NATURAL;
                     Type_ID       :    out NATURAL);

 procedure Get_PortId (Port_Name   : in     STRING;
                       Port_ID     :    out POSITIVE;
                       Queue_Bound :    out POSITIVE;
                       Data_Size   :    out NATURAL);

 procedure Get_TypeId (Type_Name   : in     STRING;
                       Type_ID     :    out NATURAL;
                       Type_Size   :    out NATURAL);

 procedure Init;
```

```
 procedure Send_Port (Port_ID   : in POSITIVE;
                      Data      : in System.Address;
                      Data_Size : in NATURAL;
                      Type_ID   : in NATURAL);

 procedure Test_Input_Port (Port_ID   : in  POSITIVE;
                   Type_of_Next_Input : out NATURAL;
                   Size_of_Next_Input : out NATURAL;
                   Inputs_in_Queue    : out NATURAL);

 procedure Test_Output_Port (Port_ID  : in  POSITIVE;
                    Spaces_Available  : out NATURAL);

 procedure Raise_Signal (Signal_Number : in NATURAL);

 procedure Safe;

end Interface;
```

References

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

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

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

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

C. G. Bell and Allen Newell, *Computer Structures: Readings and Examples*, New York: McGraw-Hill Book Company, 1971.

C. Belzile, M. Coulas, G. H. McEwen, and G. Marquis, RNET: A Hard Real Time Distributed Programming System, in *Proceedings of the 1986 Real-Time Systems Symposium*, pages 2-13, IEEE Computer Society Press, December, 1986.

Barry W. Boehm, A Spiral Model of Software Development and Enhancement, *Computer* 21(5), May, 1988.

I. Lee and S. M. Goldwasser, A Distributed Testbed for Active Sensory Processing, in *1985 International Conference on Robotics and Automation*, pages 925-930, IEEE Computer Society Press, March, 1985.

J. Magee and J. Kramer, Dynamic Configuration for Distributed Real-Time Systems, in *Proceedings of the 1983 Real-Time Systems Symposium*, pages 277-288, IEEE Computer Society Press, December, 1983.

S. A. Mamrak, H. Kuo, and D. Soni, Supporting Existing Tools in Distributed Processing Systems: The Conversion Problem, in *Proceedings of the 3rd International Conference on Distributed Computing Systems*, pages 847-853, IEEE Computer Society Press, October, 1982.

[*submitted Nov. 14, 1988; revised Jan. 19, 1989; accepted Jan. 23, 1989*]