TO APPEAR IN THE PROCEEDINGS OF THE IFIP WORKING CONFERENCE ON DECENTRALIZED SYSTEMS
LYON, FRANCE 11-13 Dec. 89

The Durra Language and Runtime Environment: Tools for PMS-Level Programming

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

Software Engineering Institute and School of Computer Science
Carnegie Mellon University, Pittsburgh, PA 15213

1. Introduction

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.* Users of heterogeneous machines 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.*

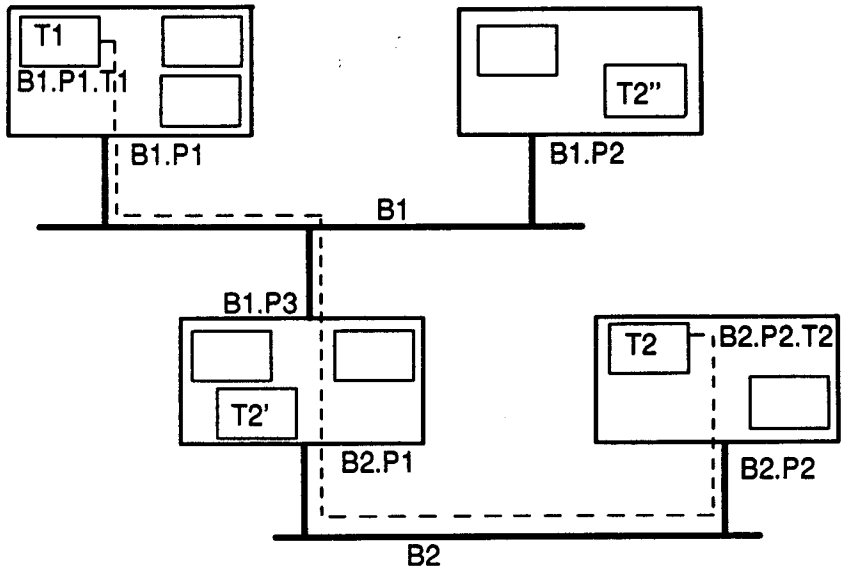

Figure 1: Binding Tasks to Resources

Currently, users of a heterogeneous machine follow the 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. As suggested in Figure 1, these programs are often written with built-in knowledge about the cooperating programs, thus making them difficult to reuse in

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.

alternative applications, or with knowledge about the network structure, making them difficult to reuse in a different environment. Tailoring the programs to the application or the environment further complicates the development of applications whose structure might change as a result of requirements of the application (e.g., mode changes in signal processing) or to support fault-tolerance (e.g., restarting a program in a different processor after the original processor fails).

It is better to separate the concerns of the developers of the individual component programs and those of the developers of the applications using these programs. We call the activities of the former group *ISP-level Programming* and the activities of the latter group *PMS-level Programming*. Developing software at the PMS-level is qualitatively different from developing software at the ISP-level. 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.

2. The Durra Language

Durra [2, 4] 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 in [5]. 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 result of compiling a Durra application description is a set of resource allocation and scheduling directives, as suggested in Figure 2.

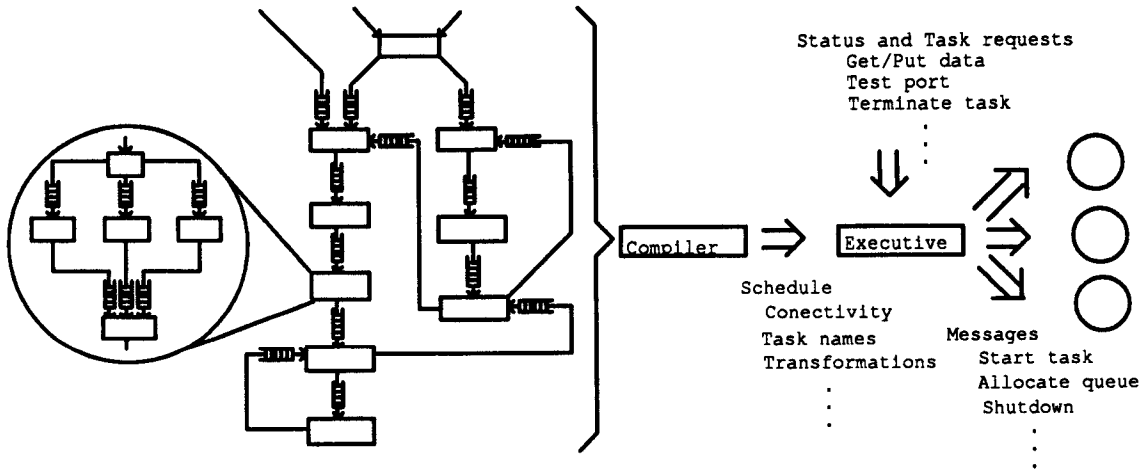

Figure 2: Compilation of a PMS-Level Program

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 3.

During the first phase, the developer writes (in the appropriate programming languages) the various tasks that will be executed as concurrent programs in the heterogeneous machine. For each of these task implementations, the developer writes (in Durra) a corresponding task description.

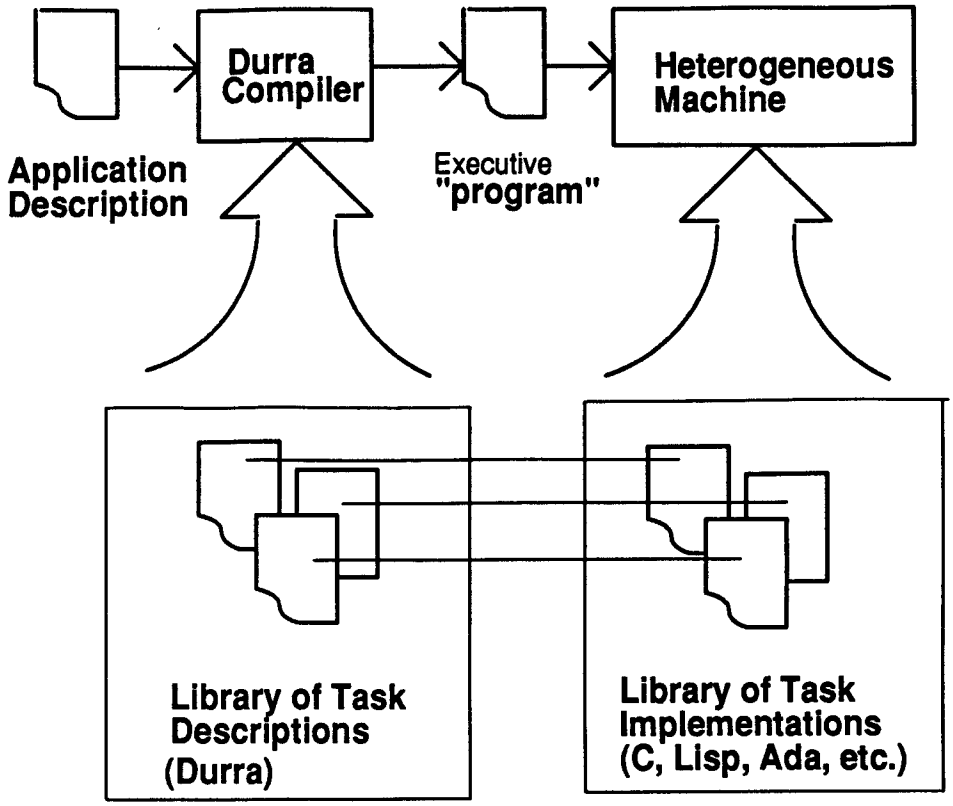

Figure 3: Developing a Durra Application

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 Ada), processor type (e.g., Motorola 68020 or DEC VAX), performance characteristics, or other properties. 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 to be interpreted by the executive.

During the last phase, the executive loads the task implementations (i.e., programs corresponding to the component tasks) into the processors and issues the appropriate commands to execute the programs.

2.2. Task Descriptions

Task descriptions are the building blocks for applications. Task descriptions include the following information (Figure 4): (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                                    -- Communication between tasks
    port-declarations

  attributes                          -- Miscellaneous properties of the task
    attribute-value-pairs

  behavior                    -- Functional and timing behavior of the task
    requires predicate
    ensures predicate
    timing timing expression

  structure                                -- Internal structure of the task
    process-declarations                   -- Instances of internal subtasks
    queue-declarations          -- Communication between internal processes
    reconfiguration-statements  -- Dynamic modifications to the structure
end task-name
```

Figure 4: A Template for Task Descriptions

Interface Information.- This portion defines the ports of the processes instantiated from the task. 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:

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

Attribute Information.- This portion specifies miscellaneous properties of a task. Attributes are a means of indicating pragmas or hints to the compiler and/or executive. 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.

We do not prescribe a mechanism for determining a timing expression. The expression could be derived by analysis of the task implementation, by observation of the task execution profile, or by other means, and then written into the task description. A timing expression is bound not only to a particular implementation (e.g., affected by the algorithm or programming language), but also to a particular execution environment (e.g., affected by the processor on which it executes). Thus, a task description that includes a timing expression, must also specify any attributes necessary to identify the exact task implementation whose behavior is given by the timing expression. For additional information about the syntax and semantics of the functional and timing behavior description, see [1].

Structural Information.- This portion defines the component tasks and queues, their interconnection, and the dynamic changes to the structure, if any.A process declaration

of the form

process_name : **task** *task_selection*

creates a process as an instance of the specified task. The task is identified by a task selection template, described later in this paper.

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 reconfiguration statement of the form

if *condition* **then**
  remove *process-and-queue-names*
  process *process-declarations*
  queues *queue-declarations*
  reconnect *queue-reconnections*
  exit *condition*
end if;

specifies changes in the current structure of the application, the conditions under which these changes take effect, and the conditions under which the changes are undone, thus reverting to a previous configuration. 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 structure. The reconfiguration and exit conditions are Boolean expressions involving time values, queue sizes, signals raised by the processes, and other information available to the executive at runtime.

2.3. Task Selections

Task selections are templates used to identify and retrieve task descriptions from the task library. A given task, e.g., "feature_detection", might have a number of different implementations that differ along dimensions such as algorithm used, code version, performance, or processor type. In order to select among a number of alternative implementations, the user provides a task selection as part of a process declaration. This task selection lists the desirable features of a suitable implementation.

```
task task-name
  ports                          -- OPTIONAL. Interface of the desired task
    port-declarations

  attributes -- OPTIONAL. Miscellaneous properties of the desired task
    attribute-expression

  behavior                       -- OPTIONAL. Behavior of the desired task
    requires predicate
    ensures predicate
    timing timing expression
end task-name                                              -- OPTIONAL.
```

Figure 5: A Template for Task Selections

Syntactically, a task selection looks somewhat like a task description without the structure part, and all other components except for the task name are optional. Figure 5 shows a template for a task selection. For brevity, if only the task name is given, the

terminating "**end** *task-name*" is optional.

There are a number of rules used to identify and select a library task, depending on the information provided in the task selection:

- Port directions and data types (but not necessarily port names) must be identical.
- Behavior of the task selection must imply the behavior of task description.
- The attribute expression (predicate) of the task selection must be "satisfied" by the attributes of task description.

The use of attributes to select a library task is illustrated in Figure 5. The task selection specifies a task name (`t`) that, in this example, matches three different task descriptions with the same name. The author and version attributes are used in an expression to specify additional properties of the desired task. In the example, exactly one task description matches all three requirements (task name, author, and version) and this is the task selected by the Durra compiler. It is an error if more than one candidate or no candidate task descriptions are left at the end of the matching operation.

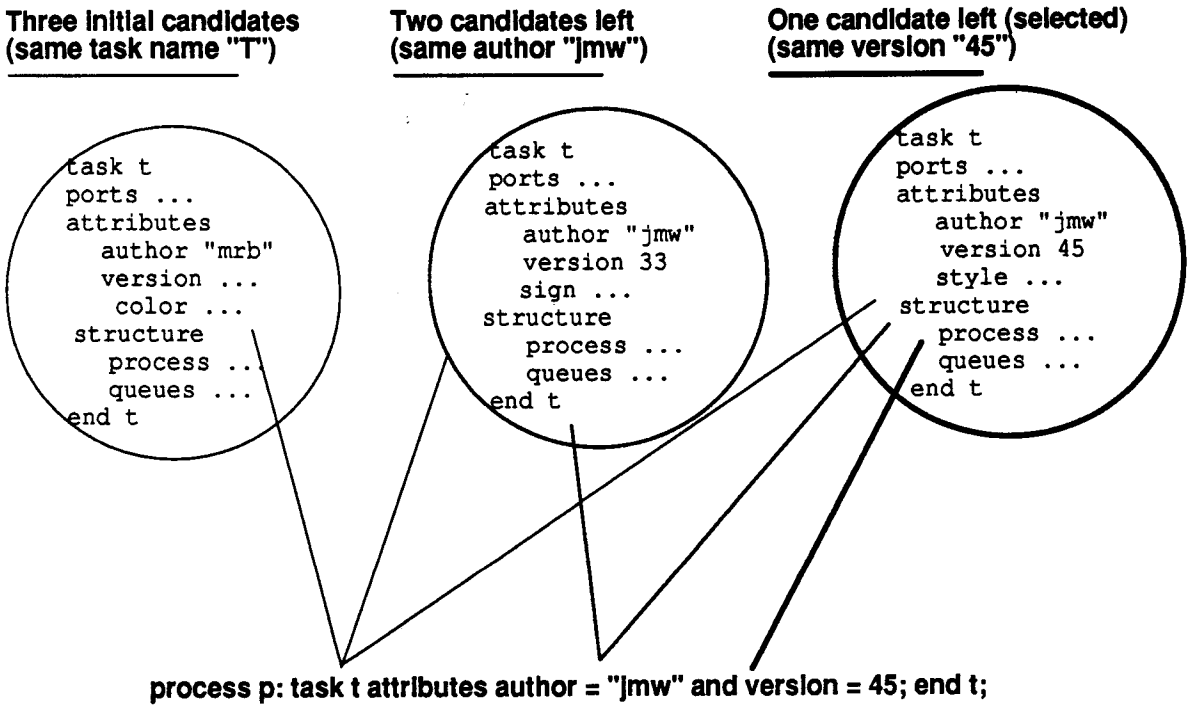

Figure 6: Selecting a Task from the Task Description Library

3. A Durra Example

The following is a complete example of a Durra application. It consists of several type declarations and component task descriptions and an application description using these types and tasks. Figure 7 shows the structure of the application.

In this application, one task (`TaskA`) sends out strings to **broadcast** task that in turn sends them on to two other tasks (`TaskB` and `TaskC`). The application description (`main`) cements all of the component tasks together. The example illustrates the use of data transformations, specified as part of a queue declaration, and the use of a predefined task, **broadcast**, which is implemented directly by the executive.

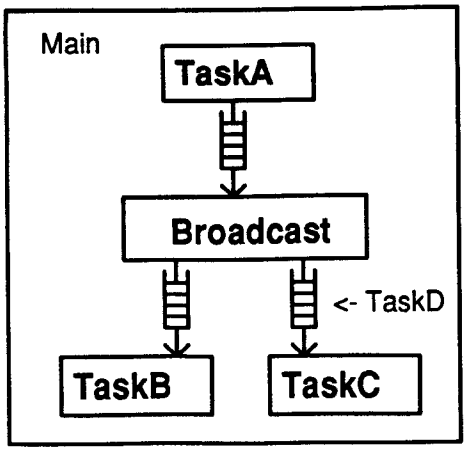

Figure 7: Durra Application Structure

Figure 8.a shows the type descriptions. `Byte` and `integer` are scalar types, 8 and 32 bits long respectively. `String` is an unbounded sequence of bytes. Figure 8.b shows the main component task descriptions. `TaskA` has a single output port, `out1`, which produces strings. It can run on any VAX processor and is implemented by the program `string_producer`. `TaskB` and `TaskC` both have a single input port, `in1`, which consume strings and integers, respectively. Figure 8.c shows two alternative task descriptions for a task, `TaskD`, that has a single input port, `in1`, which consumes strings, and a single output port `out1`, which produces integers. The two task descriptions are implemented by two different programs, `string_transformer_vax` and `string_transformer_sun`, which execute on `vax` or `sun` processors. This information is captured in the attributes of the task descriptions.

Figure 8.c, is the complete application description. It specifies the tasks that make up the application, plus an instance of the predefined task **broadcast**. The structure part specifies the interconnection of those four tasks.

One of the consumer tasks takes integers as input and thus, can not be connected directly to the output of the broadcast task. An auxilliary process `pd`, which implements some suitable string-to-integer transformation operation is specified in the queue declaration. Notice the use of attribute information in the declaration of `pd`. Of the two alternative implementations of `TaskD`, the application descriptions selects the version that will execute in the same processor as `p3`.

The declaration of queue `qb3` specifies a data transformation process `p4` that will operate on the data while they reside in `q3`. In general, a queue declaration can specify an arbitrary sequence of data transformation processes. Although these processes must be declared as the other component processes of the application, their use is syntactically simpler (the user does not need to specify intermediate queues; these are generated automatically by the compiler.)

4. Application Execution

The Durra runtime executives interpret the instructions generated by the Durra compiler. As illustrated in Figure 9, the executives can run in master or server mode. There is one server executive on each processor in the configuration and it is responsible for starting all tasks assigned to that processor. There is one master executive for the entire network and it is responsible for assigning tasks to servers, establishing communication links, and controlling the execution of the application. The executives implement the predefined tasks (**broadcast**, **merge**, and **deal**) built into the language, manage the message queues, and invoke the data transformation operations.

```
type byte is size 8;
type integer is size 32;
type string is array of byte;
```

a -- Type Descriptions

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

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

task taskc
    ports       in1: in integer;
    attributes  processor = sun;
                implementation = "integer_consumer";
end taskc;
```

b -- Component Task Descriptions

```
task taskd
    ports       in1: in string;
                out1: out integer;
    attributes  processor = vax;
                implementation = "string_transformer_vax";
end taskd;

task taskd
    ports       in1: in string;
                out1: out integer;
    attributes  processor = sun;
                implementation = "string_transformer_sun";
end taskd;
```

c -- Transformation Task Descriptions

```
task main
    structure
        process
            p1: task taska;
            p2: task taskb;
            p3: task taskc;
            p4: task taskd
                    attributes processor = p3.processor;
                end taskd;
            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 > p4 > p3.in1;
end main;
```

d -- Application Description

Figure 8: Durra Example

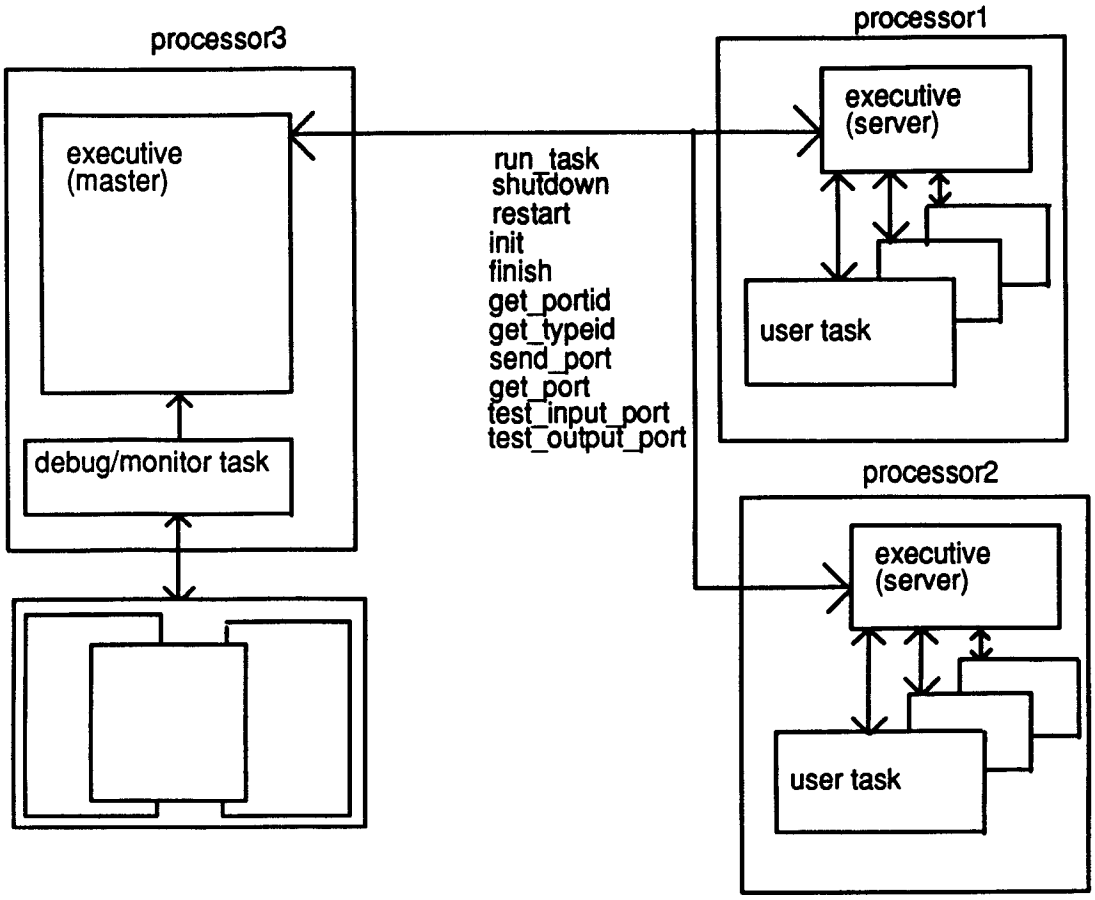

Figure 9: The Durra Runtime Environment

4.1. Resource Allocation

The user can specify, via attributes, the specific processor (or processor class) on which a process can execute. In the absence of further information, the executive attempts to balance the load on the various processors by assigning equal number of processes to each processor, ignoring the throughout requirements of the individual processes. A better policy would be to let the executives use the timing expressions characterizing the processes' behavior to determine the load imposed by each process. With this information, the executives could achieve a better processor allocation. Resource allocation takes place only at the start of the application or as a result of a reconfiguration. The executives will not reallocate resources otherwise. Admittedly, allowing the user to assign processors to processes could prevent the executives from doing a better job, with a potential loss in aplication throughput. From the onset of the project, we have been motivated by real-time, embedded applications such as avionics, robotics, and the like, in which the resources are dedicated to the application and in which predictability of behavior is sometimes more important than speed. Dynamic load balancing does not seem to be a critical issue in these domains.

4.2. Task Communications Interface

The application tasks 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.

In the current implementation the executive takes advantage of Unix communication primitives to allocate sockets for receiving remote procedures calls from the application

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 executive, 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 interface calls, application tasks typically would exhibit the following behavior:

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

4.3. Debugging and Monitoring the Application

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 executive, 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 run-time environment is the executive itself. It provides input and output ports, just like the normal processes, but these ports are used to communicate with a special Durra debugger/monitor task. The Durra debugger/monitor [8] is an interactive program which communicates with the Durra executive at runtime to provide information about and control over the progress of the application.

The debugger/monitor is an optional component of the Durra runtime environment. The user may invoke it independently at any time during the execution of a Durra application. Alternatively, the debugger/monitor may be activated as part of the runtime environment start-up procedure. As shown in Figure 9, the debugger/monitor exchanges information with the master executive through a remote procedure call (rpc) interface, just as Durra application tasks do. The debugger/monitor however, has its own distinct set of procedure calls.

Using the debugger/monitor commands, the user can

- 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 reconfigurations

5. Building Application Prototypes

To support the development of application prototypes, we have developed a program that acts as a "universal" task emulator. This program, MasterTask [3], 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 (within the precision of Durra's time values and the executive-maintained clocks).

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 for the task it wants to emulate and assigns a number of concurrent, light-weight processes (Ada task objects in the current implementation) to interpret the timing expression. These processes are responsible for evaluating the guards and for invoking the queue operations. In general, MasterTask exhibits the same input/output behavior as would the real Durra task implementation, issuing the same type of procedure calls to the executive and at the right time.

A PMS-level language like Durra provides natural support for system development methodologies based on successive refinements, such as the Spiral method [7]. Users of the spiral model selectively identify high-risk components of the product, establish their requirements, and then carry out the design, coding, and testing phases. 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.

Durra allows the designer to build mock-ups of an application, starting with a gross decomposition into tasks 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 critical to the performance of the entire system. The designers can experiment with and evaluate proposed changes in task behavior or performance by rewriting and reinterpreting the corresponding timing expression.

The designers can then proceed by replacing the original task descriptions with more detailed task descriptions, 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.

6. Conclusions

The problem of dynamic reconfiguration of distributed systems is addressed in the CONIC language and Toolkit [9, 12]. Initially CONIC restricted tasks to be programmed in a fixed language (an extension to Pascal with message passing primitives) running on homogeneous workstations but this restriction was later relaxed. RNET [6] is another language designed for 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 executives required by Durra and other languages in which the concurrent tasks are treated as black boxes.

Interfacing heterogeneous machines or language environments is not a new problem. Several techniques have been proposed to take high level specifications and generate structure/type declarations and routines which perform the appropriate packing and unpacking of the data [10, 11, 13, 14]. These and other similar facilities could be adopted by the application developers without difficulty as Durra operates at a higher level of abstraction.

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 executive, debugger/monitor, 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.

References

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

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

[3] M.R. Barbacci.
MasterTask: The Durra Task Emulator.
Technical Report CMU/SEI-88-TR-20 (DTIC AD-A199 429), Software Engineering Institute, Carnegie Mellon University, July, 1988.

[4] M.R. Barbacci and J.M. Wing.
Durra: A Task-Level Description Language Reference Manual (Version 2).
Technical Report CMU/SEI-89-TR-34, Software Engineering Institute, Carnegie Mellon University, September, 1989.

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

[6] C. Belzile, M. Coulas, G.H. MacEwen, 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.

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

[8] D.L. Doubleday.
The Durra Application Debugger/Monitor.
Technical Report CMU/SEI-89-TR-32, Software Engineering Institute, Carnegie Mellon University, September, 1989.

[9] N. Dulay, J. Kramer, J. Magee, M. Sloman, and K. Twidle.
Distributed System Construction: Experience with the Conic Toolkit.
In *Proceedings of the International Workshop on Experiences with Distributed Systems*, pages 189-212. Springer-Verlag, September, 1987.

[10] P.H. Gibbons.
A Stub Generator for Multilanguage RPC in Heterogeneous Environments.
IEEE Transactions on Software Engineering 13(1):77-87, January, 1987.

[11] M.B. Jones, R.F. Rashid, and M.R. Thompson.
Matchmaker: An Interface Specification Language for Distributed Processing.
In *Conference Record of the Twelfth Annual ACM Symposium on Principles of Programming Languages*, pages 225-235. ACM, January, 1984.

[12] 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.

[13] 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.

[14] Sun Microsystems, Inc.
XDR: External Data Representation Standard.
RFC 1014, SRI Network Information Center, June, 1987.