A Status Report on Durra: A Tool 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

Presented at the 3rd Workshop on Large-Grained Parallelism,
Software Engineering Institute, Pittsburgh, PA. October 10-11, 1989

Abstract

Durra is a language designed to support PMS-level programming. 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 application is executed under control of the runtime environment. The environment consists of a number or servers, tailored to each machine/operating system used in the network, and a scheduler that directs the initiation and termination of tasks, the transmission of data between the tasks, and the dynamic reconfiguration of the application. The scheduler performs these operations by interpreting a set of instructions generated by the compilation of the application description.

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.

1. Programming Heterogeneous Machines

A computing environment consisting of loosely-connected networks of multiple special- and general-purpose processors constitutes a *heterogeneous machine.* Users of heterogeneous machines are concerned with allocating specialized resources to concurrent, large-grained tasks. 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.*

In a heterogeneous machine, processors are not the only critical resource. In addition to special purpose processors such as array processors, and general-purpose workstations, the resources that must be allocated include communication links, data buffers with processing capabilities, etc., as illustrated in Figure 1.

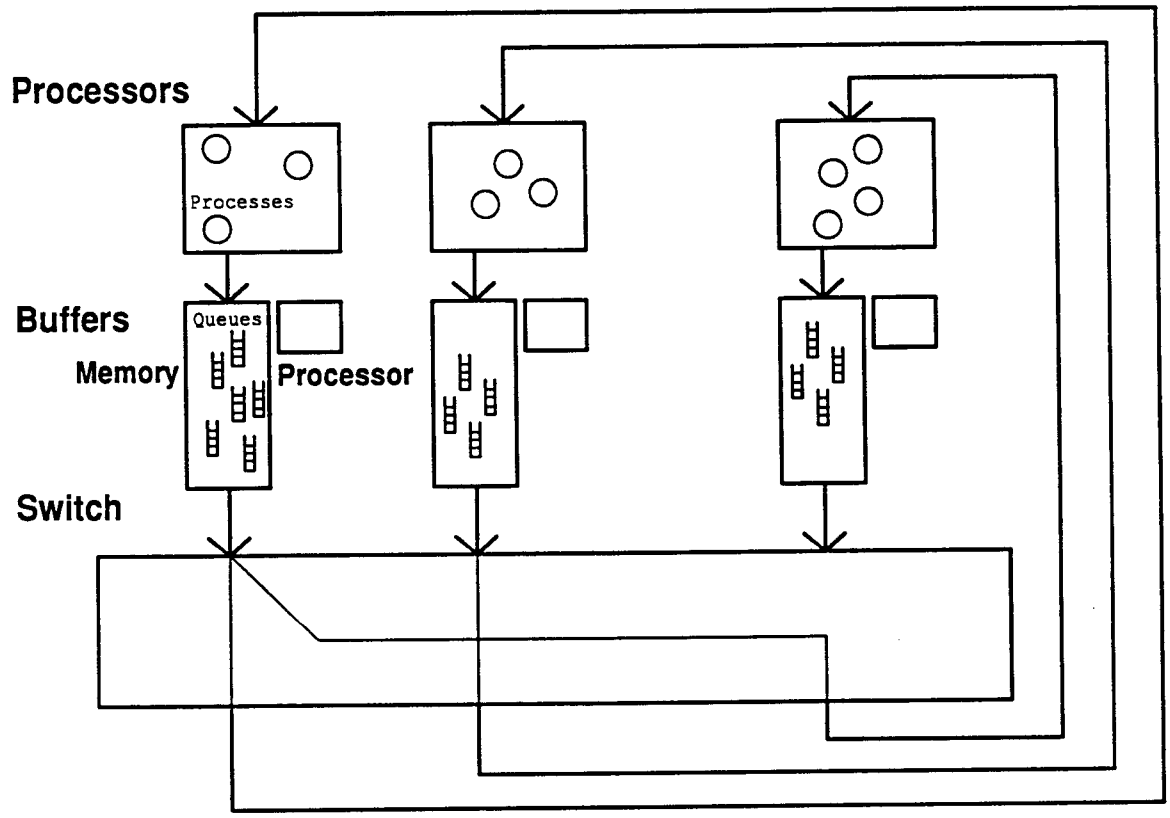

Figure 1: A Heterogeneous Machine

Current users of heterogeneous machines 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, Ada) 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. Often, these programs are written with built-in knowledge about the cooperating programs,

thus making them difficult to reuse in 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).

We believe that a better approach is 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*.

We claim that 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 report we address some of these issues by describing a language, Durra, its support tools (compiler, runtime environment, and task emulator) and describing some of the early experiments using the language and support tools.

2. The Durra Language

Durra [1, 7, 3] 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 [11].

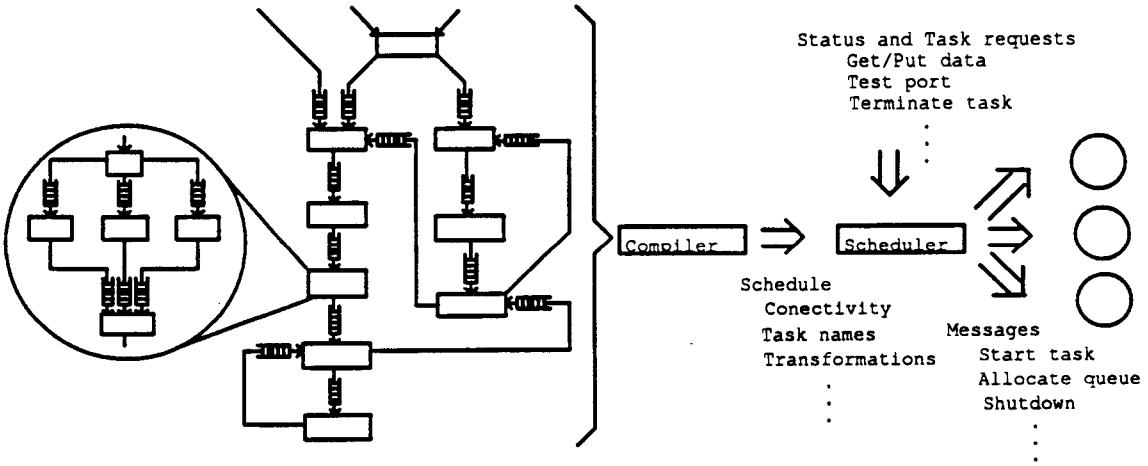

Figure 2: Compilation of a PMS-Level Program

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.

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 [2].

Structural Information.- This portion defines a process-queue graph (e.g., Figure 2) and possible dynamic reconfiguration of the graph.

A process declaration of the form

process_name : **task** *task_selection*

creates a process as an instance of the specified task.

Task selections are templates used to identify and retrieve task descriptions from the task library. A given task, e.g., convolution, might have a number of different implementations that differ along dimensions such as algorithm used, code version, performance, or processor type. In order to select among a number of alternative implementations, the user provides a task selection as part of a process declaration. 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. Functional and timing behavior of the desired task
    requires predicate
    ensures predicate
    timing timing expression
end task-name                                                    -- OPTIONAL.
```

Figure 4: 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 4 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.

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 queue declaration can specify an arbitrary sequence of data transformation processes. Although these processes must be declared as the other components of the application, their use is syntactically simpler (the user does not need to specify intermediate queues; these are generated automatically by the compiler.)

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-and-queue-names*
process *process-declarations*
queues *queue-declarations*
reconnect *queue-reconnections*
exit *condition*
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, 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 graph. The reconfiguration and exit predicates are Boolean expressions involving time values, queue sizes, signals raised by the processes, and other information available to the scheduler at runtime.

3. The Durra Runtime Environment

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

After compiling the type declarations, the component task descriptions, and the application description, the application can be executed by starting an instance of the Durra server in each processor and starting the scheduler (Section 3.1) 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.

In the remainder of this section, we sketch the components of the runtime environment: the scheduler, the servers, the debugger/monitor, and the application tasks. For additional details, see [8, 4, 9, 12].

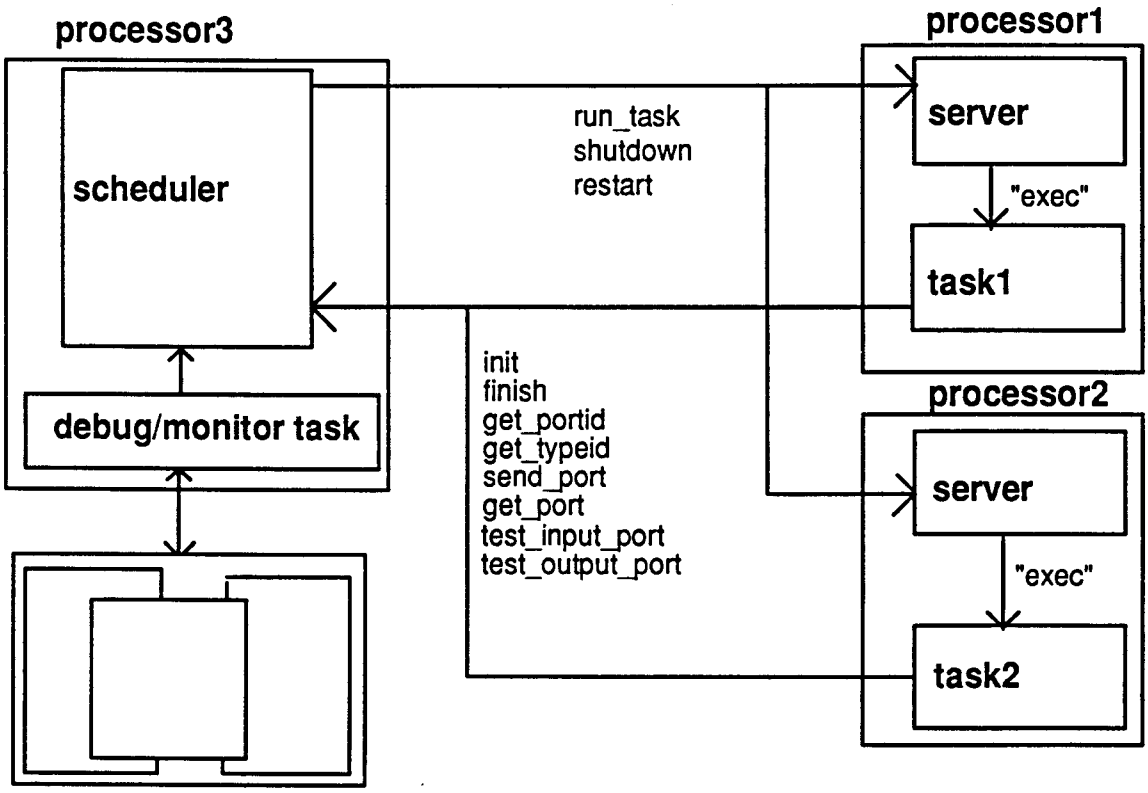

Figure 5: The Durra Runtime Environment

3.1. The Scheduler

The scheduler is the part of the Durra runtime system responsible for allocating tasks to processors, establishing communication links, and controlling the execution of the application. In addition, the scheduler implements the predefined tasks (**broadcast**, **merge**, and **deal**) and the data transformations described in [1]. 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 processes, and the possible reconfigurations to the structure of the application. These reconfigurations consist of elimination of existing processes, activation of new processes, and connection 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 supporting 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 3.4.

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. The Debugger/Monitor

The Durra application debugger/monitor is an interactive process which communicates with the Durra scheduler at runtime to provide information about and control over the progress of the application.

The focus of the monitor is at the application level rather than at the level of the individual tasks. The monitor provides the abstracted Durra view of the world, where the individual tasks are treated as black boxes connected to each other through Durra ports and queues. The user can examine scheduler-internal data, insert break points on Durra communication interfaces, change task attributes, and do other miscellaneous operations. The monitor also provides a tracking facility to monitor the flow of data through the Durra queues. Given this information, the developer can identify the performance bottlenecks and take corrective actions, such as moving a task to a faster processor or re-implementing a task to improve its efficiency.

3.4. 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 [4, 9] 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.

4. The Durra Task Emulator

In addition to the debugging facilities built into the scheduler and the debugger/monitor, a program that acts as a "universal" task emulator is available for building prototypes of applications. This program, MasterTask [5], 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; 2) evaluate a guard expression; 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 [4] for a description of the operations.)

5. Early Experiences with Durra

As Durra evolves, we have applied it to a series of example applications. Starting from simple descriptive exercises, we later demonstrated the use of Durra as a language supporting evolutionary software development. We are currently developing a distributed avionics application that stresses runtime performance and dynamic reconfiguration features.

In [6] we describe an experiment in writing task descriptions and type declarations for a subset of the Generalized Image Library, a collection of utilities developed at Carnegie Mellon University.

In [10] we describe an experiment in implementing a command, control, communications and intelligence (C^3I) node using reusable components. The experiment involved writing task descriptions and type declarations for a subset of the TRW testbed, a collection of C^3I software modules developed by TRW Defense Systems Group. The experiment illustrated the methodology for building complex, distributed systems supported by Durra. It did not, however, illustrate all the features of the language; in particular, it did not illustrate those features that support dynamic, but planned, reconfiguration of a running application, or those features supporting unplanned dynamic reconfigurations as a means to support fault tolerance.

We are currently developing a small avionics application and we hope to have it ready for demonstration at this workshop. The application consists of several concurrent tasks generating and exchanging messages with information about location, direction, and speed of the aircraft, weapon status, system faults, etc. Some of the tasks are sporadic (e.g., operator console) while others are periodic (e.g., inertial navigation, display). Early experiments show that the performance of the system is limited by the TCP/IP protocol and the Ethernet. It takes about 30 milliseconds to route a message from a source task, to the scheduler, and then to a destination task. These message-passing times come uncomfortably close to the basic task periods (10Hz for the inertial navigation task) and limit the speed of the simulated aircraft. However, we are planning on porting the Durra runtime environment to a faster architecture, developed specifically for avionics, and providing faster communications. We hope to report on this experiment at a future workshop.

6. Conclusions

PMS-level programming, as implemented by Durra, lifts the level of programming at the code level (task implementations) to programming at the specification level (task descriptions), separating the structure of an application from its behavior. This separation provides users with control over the evolution of an application during application development as well as during application execution. During development, an application evolves as the requirements of the application are better understood or change. This evolution takes the form of changes in the application description, modifying task selection templates to retrieve alternative task implementations from the library, and connecting these implementations in different ways to reflect alternative designs. During execution, an application evolves through application mode changes or in response to faults in the system. This evolution takes the form of conditional, dynamic reconfigurations, removing processes and queues, instantiating new processes and queues, and building a new process-queue graph without affecting the remaining processes and queues.

The PMS-level programming paradigm fits very naturally a top-down, incremental method 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 development of large, distributed systems. 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. In the process of developing the application, component tasks can be decomposed into simpler process-queue graphs and at each stage of the process, the application can be emulated using MasterTask as a stand-in for the yet-to-be written task implementations.

In our prototype implementation, we have intentionally sacrificed semantic complexity in favor of simpler task selection based only on interface and attribute information, and have limited the performance and reliability by implementing a centralized scheduler. As a result, we gained the advantage of being able immediately to test our general idea about PMS-level programming with a real environment (Durra compiler, runtime system, and debugger/monitor) that runs on a heterogeneous machine (various kinds of workstations connected via an Ethernet). Expanding task selection features and distributing the runtime scheduler are in our plans for the future.

References

[1] M.R. Barbacci and J.M. Wing.
Durra: A Task-Level Description Language.
Technical Report CMU/SEI-86-TR-3 (DTIC AD-A178 975), Software Engineering Institute, Carnegie Mellon University, December, 1986.

[2] 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.

[3] 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.

[4] M.R. Barbacci, D.L. Doubleday, and C.B. Weinstock.
The Durra Runtime Environment.
Technical Report CMU/SEI-88-TR-18 (DTIC AD-A199 480), Software Engineering Institute, Carnegie Mellon University, July, 1988.

[5] 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.

[6] M.R. Barbacci and D.L. Doubleday.
Generalized Image Library: A Durra Application Example.
Technical Report CMU/SEI-88-TR-19 (DTIC AD-A199 481), Software Engineering Institute, Carnegie Mellon University, July, 1988.

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

[8] M.R. Barbacci, D.L. Doubleday, and C.B. Weinstock.
Durra: A Task-Level Description Language User's Manual.
Technical Report CMU/SEI-89-TR-33, Software Engineering Institute, Carnegie Mellon University, September, 1989.

[9] M.R. Barbacci, D.L. Doubleday, C.B. Weinstock, and J.M. Wing.
Developing Applications for Heterogeneous Machine Networks: The Durra Environment.
Computing Systems 2(1), March, 1989.

[10] M.R. Barbacci, D.L. Doubleday, C.B. Weinstock, S.L. Baur, D.C. Bixler, M.T. Heins.
Command, Control, Communications, and Intelligence Node: A Durra Application Example.
Technical Report CMU/SEI-89-TR-9 (DTIC AD-A206575), Software Engineering Institute, Carnegie Mellon University, February, 1989.

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

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

Third Workshop on Large Grain Parallelism

Software Engineering Institute

October 10-11, 1989

Carnegie Mellon University
Pittsburgh, Pennsylvania

Sponsored by the
U.S. Department of Defense