A Language for Distributed Applications

Mario R. Barbacci and Jeannette M. Wing

Software Engineering Institute and School of Computer Science
Carnegie Mellon University

Abstract

Durra is a language designed to support the development of distributed applications consisting of multiple, concurrent, large-grained tasks executing in a heterogeneous network.

An application-level program is written in Durra as a set of *task descriptions* 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 intermediate queues required to store the messages as they move from producer to consumer processes, and the possible dynamic reconfigurations of the application.

The application-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.

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 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 *application-level programming*, to distinguish them from the activities leading to the development of the individual component tasks.

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.

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, 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 from those of the developers of the applications using these programs. We claim that developing software in this style is qualitatively different from developing software at the level of the component programs. 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 showing how it can support a top-down, incremental software development methodology. In this paper we only address language and methodology issues. For a description of the Durra runtime environment and tools see [4].

2. The Durra Language

Durra [2, 5] is a language designed to support the development of distributed applications. An application is written in Durra as a set of *task descriptions* that prescribes a way to manage the resources of a heterogeneous machine network. The result of compiling a Durra application description is a set of resource allocation and scheduling directives, as suggested in Figure 1.

CH2854-8/90/0000/0059$01.00 © 1990 IEEE

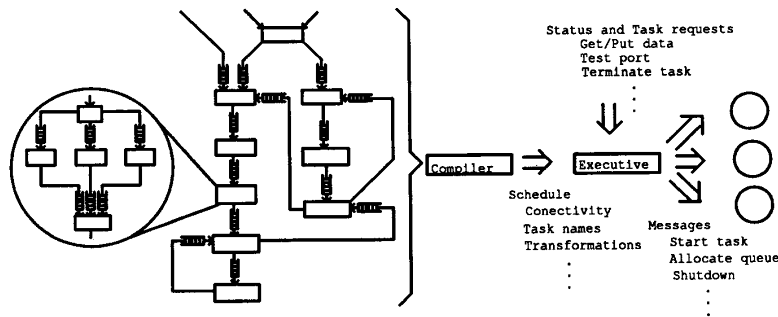

Figure 1: Compilation of an Application Description

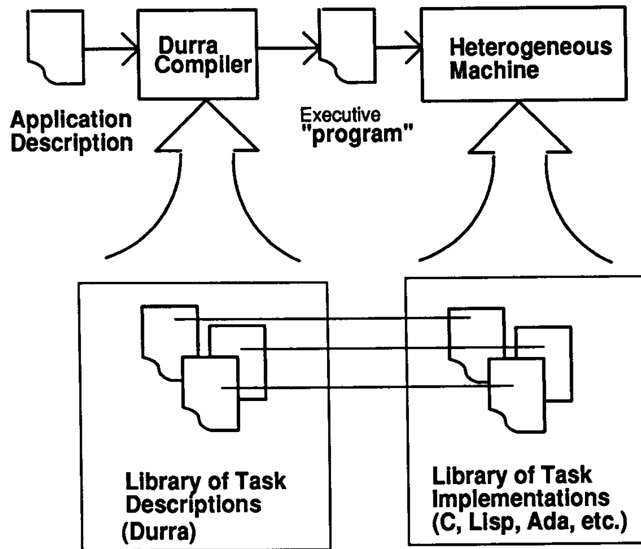

Figure 2: Scenario for Developing a Distributed Application

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 2.

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.

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.

```
task task-name                                    -- Name of the task or task family
  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

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 commands to be interpreted by the executive.

During the last phase, the executive loads the task implementations (i.e., programs corresponding to the component task descriptions) 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 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.

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. 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 [1].

Structural Information.- This portion defines a process-queue graph (e.g., Figure 1) 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. Syntactically, a task selection looks somewhat like a task description without the structure part. Figure 4 shows a template for a task selection.

A queue declaration of the form

queue_name [*queue_size*]:
src_port > *data_transformation* > *dst_port*

```
task task-name                               -- REQUIRED. Name of a task or task family.
  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                                        -- OPTIONAL.
end task-name
```

Figure 4: A Template for Task Selections

creates a queue through which data flow from an output port of a process (src_port) into the input port of another process (dst_port). 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;
```

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 conditions are Boolean expressions involving time values, queue sizes, signals raised by the processes, and other information available to the executive at runtime.

3. A Task Emulator as a Prototyping Tool

To support the prototyping of distributed, large-grained applications, 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 input and output in the proper sequence and at the proper time (within the precission of Durra time values and the executive clock.)

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.

3.1. Timing Expressions

Timing expressions are the critical piece of information used by MasterTask. Tasks send and receive messages following a task-specific pattern provided by a timing expression. This expression describes the behavior of the task in terms of the operations it performs on its input and output ports; this is the behavior of the task seen from the outside.

Queue operations constitute the basic events of a timing expression. An event represents a queue operation (i.e., "Enqueue" and "Dequeue") on the queue attached to a specific port. In addition, a pseudo-operation, "delay", is used to represent the time consumed between (real) queue operations.

A timing expression (Figure 5) is a regular expression describing the patterns of execution of operations on the input and output ports of a task. The optional keyword **loop** can be used to indicate that the pattern of operations is repeated indefinitely.

A timing expression is a sequence of parallel event expressions. Each parallel event expression consists of one or more event expressions separated by the symbol || to indicate that their executions overlap. Since the expressions might take different amounts of time to complete, nothing can be said about their completion, other than a parallel event expression terminates when the last event terminates.

A basic event expression is either a queue operation (including "delay") or a timing expression enclosed in parentheses. The latter form also allows for the specification of a guard, an expression specifying the conditions under which a sequence of operations is allowed to start or repeat its execution.

When MasterTask starts, it reads the 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.

Syntax:

```
TimingExpression   ::= {''LOOP''} SequentialEvent
SequentialEvent    ::= ParallelEvent_List_spaces
ParallelEvent      ::= BasicEvent_List_double_vertical_bar
BasicEvent         ::= Event |
                       {Guard ''=>''} ''('' SequentialEvent '')''
Event              ::= PortName |
                       ''DELAY'' TimeWindow
TimeWindow         ::= ''['' TimeValue '','' TimeValue '']''
TimeValue          ::= Number_Of_Seconds {TimeBase} |
                       ''*''                  -- Indeterminate amount of time
TimeBase           ::= ''DTIME'' |                 -- Time since start of day
                       ''ATIME'' |           -- Time since start of application
                       ''PTIME'' |               -- Time since start of process
Guard              ::= ''REPEAT'' IntegerValue |
                       ''BEFORE'' TimeValue |                   -- Absolute time
                       ''AFTER'' TimeValue |                    -- Absolute time
                       ''DURING'' TimeWindow |                  -- Time interval
                       ''WHEN'' Expression              -- A Boolean expression
```

Examples:

```
3615.5 atime          -- An application relative time: 1 hour and 15.5 seconds
                                     -- after the start of the application.

delay[10, 15]           -- A delay interval lasting between 10 and 15 seconds.

delay[*, 10]                    -- A delay interval taking at most 10 seconds.

delay[10, *]                   -- A delay interval taking at least 10 seconds.

in1 || in2           -- Two parallel input operations, starting simultaneously.

in1 delay[10,15] out1                            -- Three sequential operations.

repeat 5 => (in1 delay[10,15] out1)
                     -- Same as above but as a cycle repeated five times.
before 64800 DTIME => ( . . . )
                           -- A sequence constrained to start before 6 pm.
                               (64,800 seconds after the start of the day)
after 64800 DTIME => ( . . . )
                            -- A sequence constrained to start after 6 pm.
during [40.5 PTIME, 100] => ( . . . )
       -- A sequence constrained to start between 40.5 and 140.5 seconds
                                           -- from the start of the process.
when (Current_Size(in1) > 0) and (Current_Size(in2) > 0) => ((in1 || in2) out1);
                 -- A sequence that starts after both input queues have data.
loop when (Current_Size(in1) > 0) and (Current_Size(in2) > 0) => ((in1 || in2) out1);
                      -- The same sequence as above but repeated indefinitely.
```

Figure 5: Timing Expressions

3.2. Using the Task Emulator

The task emulator described above provides natural support for system development methodologies based on successive refinements, such as the Spiral method [8]. 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 likely to affect the performance of the entire system.

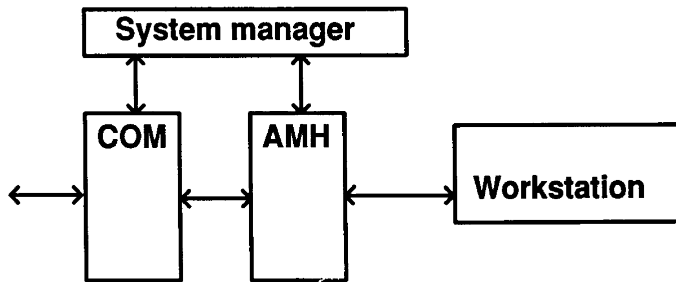

Figure 6: Initial Structure of the C^3I Node

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.

4. An Example of Incremental Development

To illustrate an incremental development process using Durra, in this section we show an application, a C^3I node [6]. The top level structure of the node is shown in Figure 6. The node consists of four subsystems: System Manager, Communications (COM), Application Message Handler (AMH), and Workstation. These subsystems correspond to the first four process declarations in Figure 7. In addition to these task, there are two auxiliary tasks which are used for communications between the system manager and the communication and application message handler subsystems.

One of these auxiliary tasks broadcasts commands from the system manager to the other two subsystems, and the other merges their responses to the system manager. **Broadcast** and **merge** are predefined in Durra and implemented directly by the Durra runtime executive. A broadcast task takes data from a single input port and copies it to multiple output ports (the number of output ports is specified in the task selection.) A merge task takes data from multiple input ports and copies them into a single output port (the number of input ports is specified in the task selection.)

Figure 8 shows the tasks descriptions for the subsystems of the initial configuration. At the start of the development process we might not be ready to commit to any particular structure for the subsystems and simply opt to describe them as simple, unstructured tasks. This information is sufficient to do static checks, including port (i.e., type) compatibility and graph connectivity. However, if we want to carry out some preliminary dynamic checks, we need to provide a pseudo-implementation for each subsystem. That is, we need to write ad hoc programs that emulate the input/output behavior of each of the subsystems and then specify these programs as the "implementation" attributes in the subsystem task descriptions.

Alternatively, if a subsystem's behavior is relatively simple and repetitive, we could use MasterTask as a subsystem emulator by specifying "master" as its "implementation" attribute. In fact, we can mix the two approaches and have some subsystems emulated by ad hoc programs, while other subsystems are emulated via MasterTask, as illustrated in Figure 8.

After some experimentation with the gross decomposition outlined above, we can proceed to expand the subsystems. For example, the Message Handler subsystem (Figure 9) consists of five internal tasks. Three of these tasks, `AMHS_control`, `AMHS_inbound`, `AMHS_outbound`, are user-implemented. The other two tasks are instances of the predefined broadcast and merge tasks described before.

```
task configuration
structure
    process
      --    real system processes
        sm      : task system_manager;
        com     : task comm;
        amh     : task amhs;
        wp      : task wkstn;
      --    auxiliary system processes
        bc  : task broadcast              -- command broadcast
                ports
                        in1          : in  system_command;
                        out1, out2 : out system_command;
              end broadcast;
        mg  : task merge                  -- response multiplexor
                ports
                        in1, in2     : in  subsystem_response;
                        out1         : out subsystem_response;
                attribute
                        mode = fifo;
              end merge;
    queues
        -- system command propagation
        q_c1    : sm.SM_Out            >> bc.in1;
        q_c2    : bc.out1              >> com.SM_Commands;
        q_c3    : bc.out2              >> amh.SM_Commands;
        -- subsystem response propagation
        q_r1    : com.SM_Responses  >> mg.in1;
        q_r2    : amh.SM_Responses  >> mg.in2;
        q_r3    : mg.out1              >> sm.SM_In;
        -- inbound message propagation
        q_i1    : com.Inbound          >> amh.COMM_Inbound;
        q_i2    : amh.WS_Inbound       >> wp.Inbound;
        -- outbound message propagation
        q_o1    : wp.Outbound          >> amh.WS_Outbound;
        q_o4    : amh.COMM_Outbound >> com.Outbound;
end configuration;
```

Figure 7: Initial Application Description

The subsystem is an abstraction and does not correspond to an executable program. Its ports (`SM_Commands`, `SM_Response`, `COMM_Inbound`, `COMM_Outbound`, `WS_Inbound`, and `WS_Outbound`) must be implemented by internal-process ports. This is the purpose of the **bind** declarations, which declare which internal-process ports implement the subsystem ports.

The development of the Message Handler does not necessarily stop here. Each of the three user-implemented tasks (`AMHS_Control`, `AMHS_inbound`, and `AMHS_outbound`) could in turn consist of multiple, concurrent internal programs. The task description for `AMHS_Control`, for example, would declare internal processes and queues, and would bind internal ports to implement the interface of the task (i.e., the `SM_In`, `SM_Out`, `Cmd_Out`, and `Resp_In` ports.) This level of detail is not visible in the description of Figure 9.

We can continue the design in this fashion, successively refining the subsystem descriptions until, at the end, the application is fully described as a hierarchical graph in which the innermost nodes are implemented as separate programs, specified by the "implementation" attribute of the corresponding task descriptions.

An application description does not use language features beyond those used in a compound task description. An application description is simply a compound task description which is compiled and stored in a Durra library and, conceivably, could be used as a building block for a larger application. From the point of view of the users of Durra, the main difference between a task description and an application description is that application descriptions are translated into directives to the runtime executive by executing an optional *code generation phase* of the Durra compiler.

For brevity, we will not describe the complete design process of the C^3I node. See [7] for details about the Durra task descriptions and the task implementations.

5. Related Work

CONIC [12] address the problem of dynamic reconfiguration of real-time systems in the design of the CONIC language. Originally, CONIC restricted tasks to be programmed in a fixed language (an extension to Pascal with message passing primitives) running on homogeneous workstations. This restriction was later relaxed to support multiple programming languages.

System Manager Subsystem

```
task system_manager
    ports        SM_In    : in  subsystem_response;
                 SM_Out   : out system_command;
    attributes   implementation = "system_manager_emulator";
                 processor = "Vax";
end system_manager;
```

Communications Subsystem

```
task comm
    ports        SM_Commands     : in  system_command;
                 SM_Responses    : out subsystem_response;
                 Inbound         : out comm_if_message;
                 Outbound        : in  comm_if_message;
    attributes   implementation = "comm_emulator";
                 processor = "Vax";
end comm;
```

Application Message Handler Subsystem

```
task AMHS
    ports        SM_Commands     : in  system_command;
                 SM_Responses    : out subsystem_response;
                 COMM_Inbound    : in  comm_if_message;
                 COMM_Outbound   : out comm_if_message;
                 WS_Inbound      : out workstation_if_message;
                 WS_Outbound     : in  comm_if_message;
    attributes   implementation = "amhs_emulator";
                 processor = "Vax";
end AMHS;
```

Workstation Subsystem

```
task wkstn
    ports        Inbound         : in  workstation_if_message;
                 Outbound        : out comm_if_message;
    behavior
        timing loop ( (Inbound delay[5, 60]) || (delay[*, 180] Outbound) );
    attributes   implementation = "master";
                 processor = "Vax";
end wkstn;
```

Figure 8: Top Level Subsystem Descriptions

MINION [MINION89] consists of a language for describing distributed applications and a graphics editor for interactive modification of the application structure. MINION allows a user to expand, contract, or reconfigure an application in arbitrary ways during execution time. Hermes [1] hides from the programmers all knowledge about storage layout, persistency of objects or even operating system primitives. Processes communicate through ports, connected via message queues although the semantic of queue operations are similar to an Ada entry call/accept mechanisms, albeit the binding of processes to ports is dynamic, as in Durra, CONIC, and MINION.

RNET [8] is language 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 runtime executived required by Durra and other languages in which the concurrent tasks are treated as black boxes.

Specifying a data transformation in a queue declaration is a way to support the transmission of structured data types between heterogeneous processors or languages. For example, two tasks written in different languages are likely to use different layouts for record data types. The Durra runtime executive will not alter the presentation of the data to hide the layout differences and will transmit a record type as a block of bytes, without attempting to modify the data. It is up to the source and destination tasks to implement the appropriate packing and unpacking of the data.

Interfacing heterogeneous machines or language environments is not a new problem. Several techniques have been proposed to generate 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 in the data transformation tasks or in the application tasks proper.

```
task AMHS
    ports       SM_Commands      : in  system_command;
                SM_Responses     : out subsystem_response;
                COMM_Inbound     : in  comm_if_message;
                COMM_Outbound    : out comm_if_message;
                WS_Inbound       : out workstation_if_message;
                WS_Outbound      : in  comm_if_message;
    structure
        process ac: task AMHS_control;
                ai: task AMHS_inbound;
                ao: task AMHS_outbound;
                pb: task broadcast
                        port    in1          : in  system_command;
                                out1, out2 : out system_command;
                    end broadcast;
                pm: task merge
                        port    in1, in2 : in  subsystem_response;
                                out1     : out subsystem_response;
                        attribute mode = fifo;
                    end merge;
        bind    SM_Commands      = ac.SM_In;
                SM_Responses     = ac.SM_Out;
                COMM_Inbound     = ai.COMM_Inbound;
                COMM_Outbound    = ao.COMM_Outbound;
                WS_Inbound       = ai.WS_Inbound;
                WS_Outbound      = ao.WS_Outbound;
        queue   q1: ac.Cmd_Out   >> pb.in1;
                q2: pb.out1      >> ai.Cmd_In;
                q3: pb.out2      >> ao.Cmd_In;
                q4: ai.Resp_Out >> pm.in1;
                q5: ao.Resp_Out >> pm.in2;
                q6: pm.out1      >> ac.Resp_In;
end AMHS;
```

Figure 9: Message Handler Subsystem Description

6. Conclusions

Application-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, selecting 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 application-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 executive. As a result, we gained the advantage of being able immediately to test our general idea about application-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 executive are in our plans for the future.

References

[1] D.F. Bacon, R.E. Strom, and S.A. Yemini.
Hermes User Manual.
Technical Report, IBM Thomas J. Watson Research Center, 1988.

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

[5] 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.

[6] 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.

[7] 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.

[8] 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.

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

[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. Kramer and J. Magee.
A Model for Change Management.
In *Proceedings of the IEEE Workshop on Trends for Distributed Computing Systems in the 1990's*, pages 286-295. IEEE Computer Society, September, 1988.

[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.