Technical Report

CMU/SEI-89-TR-009
ESD-TR-89-017

Command, Control, Communications, and Intelligence Node: A Durra Application Example

Mario R. Barbacci
Dennis L. Doubleday
Charles B. Weinstock

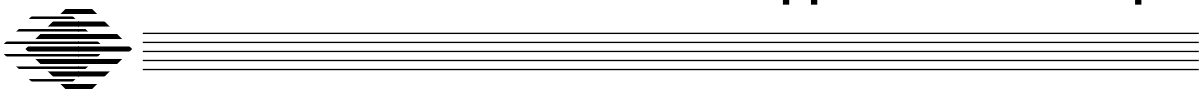
Steven L. Baur
David C. Bixler
Michael T. Heins

February 1989

Technical Report

CMU/SEI-89-TR-009
ESD-TR-89-017
February 1989

Command, Control, Communications, and Intelligence Node: A Durra Application Example



Mario R. Barbacci
Dennis L. Doubleday
Charles B. Weinstock

Software for Heterogeneous Machines Project
Software Engineering Institute

Steven L. Baur
David C. Bixler
Michael T. Heins

Reusable C^3I Node Project
TRW Defense Systems Group

Software Engineering Institute
Carnegie Mellon University
Pittsburgh, Pennsylvania 15213

This report was prepared for the SEI Joint Program Office HQ ESC/AXS

5 Eglin Street

Hanscom AFB, MA 01731-2116

The ideas and findings in this report should not be construed as an official DoD position. It is published in the interest of scientific and technical information exchange.

FOR THE COMMANDER

(signature on file)

Thomas R. Miller, Lt Col, USAF, SEI Joint Program Office

This work is sponsored by the U.S. Department of Defense.

This work was created in the performance of Federal Government Contract Number F19628-95-C-0003 with Carnegie Mellon University for the operation of the Software Engineering Institute, a federally funded research and development center. The Government of the United States has a royalty-free government-purpose license to use, duplicate, or disclose the work, in whole or in part and in any manner, and to have or permit others to do so, for government purposes pursuant to the copyright license under the clause at 52.227-7013.

This document is available through Asset Source for Software Engineering Technology (ASSET) / 1350 Earl L. Core Road ; P.O. Box 3305 / Morgantown, West Virginia 26505 / Phone: (304) 284-9000 / Fax: (304) 284-9001 / e-mail: sei@asset.com / WWW: http://www.asset.com/sei.html

Copies of this document are available through the National Technical Information Service (NTIS). For information on ordering, please contact NTIS directly: National Technical Information Service / U.S. Department of Commerce / Springfield, VA 22161. Phone: (703) 487-4600.

This document is also available through the Defense Technical Information Center (DTIC). DTIC provides acess to and transfer of scientific and technical information for DoD personnel, DoD contractors and potential con tractors, and other U.S. Government agency personnel and their contractors. To obtain a copy, please contact DTIC directly: Defense Technical Information Center / 8725 John J. Kingman Road / Suite 0944 / Ft. Belvoir, VA 22060-6218. Phone: 1-800-225-3842 or 703-767-8222.

Command, Control, Communications, and Intelligence Node: A Durra Application Example

Abstract: Durra is a language designed to support the construction of distributed applications using concurrent, coarse-grain tasks running on networks of heterogeneous processors. An application written in Durra 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.

This report describes an experiment in implementing a command, control, communications and intelligence (C^3I) node using reusable components. The experiment involves 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 illustrates the development of a typical Durra application. This is a three-step process: first, a collection of tasks (programs) is designed and implemented (these are the testbed programs); second, a collection of task descriptions corresponding to the task implementations is written in Durra, compiled, and stored in a library; and finally, an application description is written in Durra and compiled, resulting in a set of resource allocation and scheduling commands to be interpreted at runtime.

This report illustrates the methodology for building complex, distributed systems supported by Durra. It does not, however, illustrate all the features of the language; in particular, it does 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. These considerations are the subject of current design and development and will be the subject of a future report.

1. Introduction to Durra

Durra [1, 2] is a language designed to support the construction of distributed applications using concurrent, coarse-grained tasks running on networks of heterogeneous processors. An application written in Durra selects and reuses *task descriptions* and *type declarations* stored in a library. The application describes the tasks to be instantiated and executed as concurrent processes, the types of data to be exchanged by the processes, and the intermediate queues required to store the data as they move from producer to consumer processes.

Because tasks are the primary building blocks, we refer to Durra as a *task-level description language*. We use the term ‘‘description language’’ rather than ‘‘programming language’’ to emphasize that a Durra application is not translated into object code in some kind of executable (conventional) ‘‘machine language.’’ Instead, a Durra application is a description of the structure and behavior of a logical machine to be synthesized into resource allocation

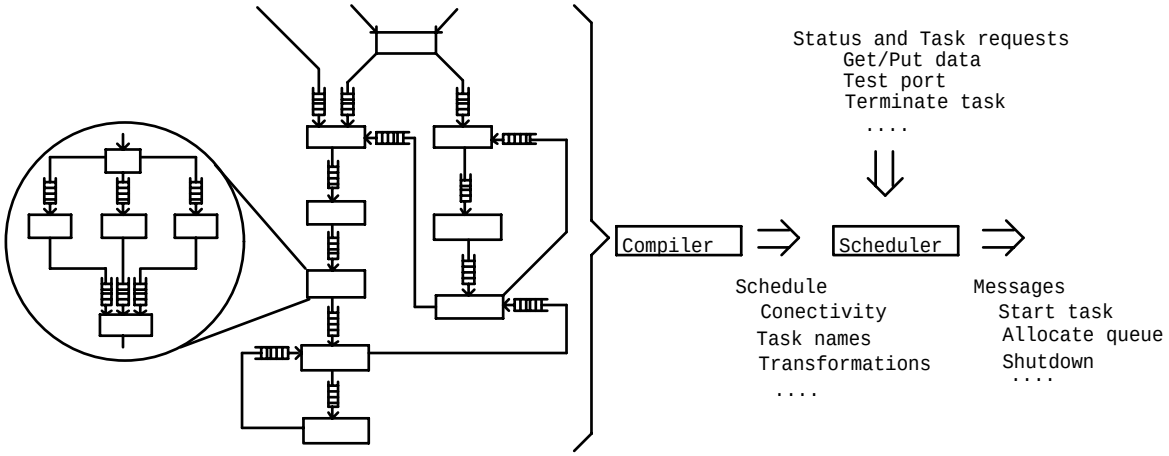


Figure 1-1: Compilation of an Application Description

and scheduling directives that are then interpreted by a combination of software, firmware, and hardware in each of the processors and buffers of a heterogeneous machine. This is the translation process depicted in Figure 1-1.

1.1. Type Declarations

The data types transmitted between the tasks are declared independently of the tasks. In Durra, these data type declarations specify scalars (of possible variable length), arrays, simple record types, or unions of other types, as shown in Figure 1-2.

```
type packet is size 128 to 1024;
                                        -- Packets are of variable length.
type tails is array (5 10) of packet;
                                    -- Tails are 5 by 10 arrays of packets.
type rec is record (rows: integer, columns: integer, data: packet);
                            -- Rec data consists of two integers and a packet.
type mix is union (heads, tails);
                                      -- Mix data could be heads or tails.
```

Figure 1-2: Durra Type Declarations

1.2. Task Descriptions

Task descriptions are the building blocks for applications. A task description includes the following information (Figure 1-3): (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 task functional and timing behavior
      functional specification
      timing specification

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

Figure 1-3: A Template for Task Descriptions

The interface information declares 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;
```

The attribute information 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;
```

The behavioral information specifies functional and timing properties about the task. The functional information part of a task description consists of a pre-condition on what is required to be true of the data coming through the input ports, and a post-condition on what is guaranteed to be true of the data going out through the output ports. The timing expression describes the behavior of the task in terms of the operations it performs on its input and output ports. For additional information about the syntax and semantics of the functional and timing behavior description, see the Durra reference manual [1].

The structural information defines a process-queue graph (e.g., Figure 1-1) and possible dynamic reconfiguration of the graph as shown in Figure 1-4.

```
task comm
   -- red communications processing
ports
        SM_Commands     : in  system_command;
        SM_Responses    : out subsystem_response;
        Inbound         : out comm_if_message;
        Outbound        : in  comm_if_message;
structure
        process
                cc      : task comm_control;
                ci      : task comm_inbound;
                co      : task comm_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     = cc.SM_In;
                SM_Responses    = cc.SM_Out;
                Inbound         = ci.Inbound;
                Outbound        = co.Outbound;
        queue
                q1      : cc.Cmd_Out    >> pb.in1;
                q2      : pb.out1       >> ci.Cmd_In;
                q3      : pb.out2       >> co.Cmd_In;
                q4      : ci.Resp_Out   >> pm.in1;
                q5      : co.Resp_Out   >> pm.in2;
                q6      : pm.out1       >> cc.Resp_In;
                q7      : co.Echo_Out   >> ci.Echo_In;
end comm;
```

Figure 1-4: Compound Task Description

A process declaration of the form

process_name **: task** *task_selection*

creates a process as an instance of the specified task. Since a given task (e.g., convolution) might have a number of different implementations that differ along different dimensions such as algorithm used, code version, performance, or processor type, the task selection in a process declaration specifies the desirable features of a suitable implementation. The presence of task selections within task descriptions provides direct linguistic support for hierarchically structured tasks.

A queue declaration of the form

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

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

A port binding of the form

task_port **=** *process_port*

maps a port on an internal process to a port defining the external interface of a compound task.

Although not illustrated in the figure, Durra provides a reconfiguration statement of the form

if *condition* **then**
    remove *process-names*
    process *process-declarations*
    queues *queue-declarations*
end if;

as a directive to the scheduler. It is used to specify changes in the current structure of the application (i.e., process-queue graph) and the conditions under which these changes take effect. Typically, a number of existing processes and queues are replaced by new processes and queues, which are then connected to the remainder of the original graph. The reconfiguration predicate is a Boolean expression involving time values, queue sizes, and other information available to the scheduler at runtime.

1.3. Scenario

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 1-5.

During the first phase, the developer writes descriptions of the component tasks and declarations of the data types. The type declarations specify the kinds of data that will be produced and consumed by the tasks in the application. The task descriptions specify the properties of the task implementations (programs). For a given task, there may be many implementations, differing in programming language (e.g., C or assembly language), processor type (e.g., Motorola 68020 or IBM 1401), performance characteristics, or other attributes. For each implementation of a task, a description must be written in Durra, compiled, and entered in the library.

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

During the last phase, the scheduler loads the task implementations (i.e., programs cor-

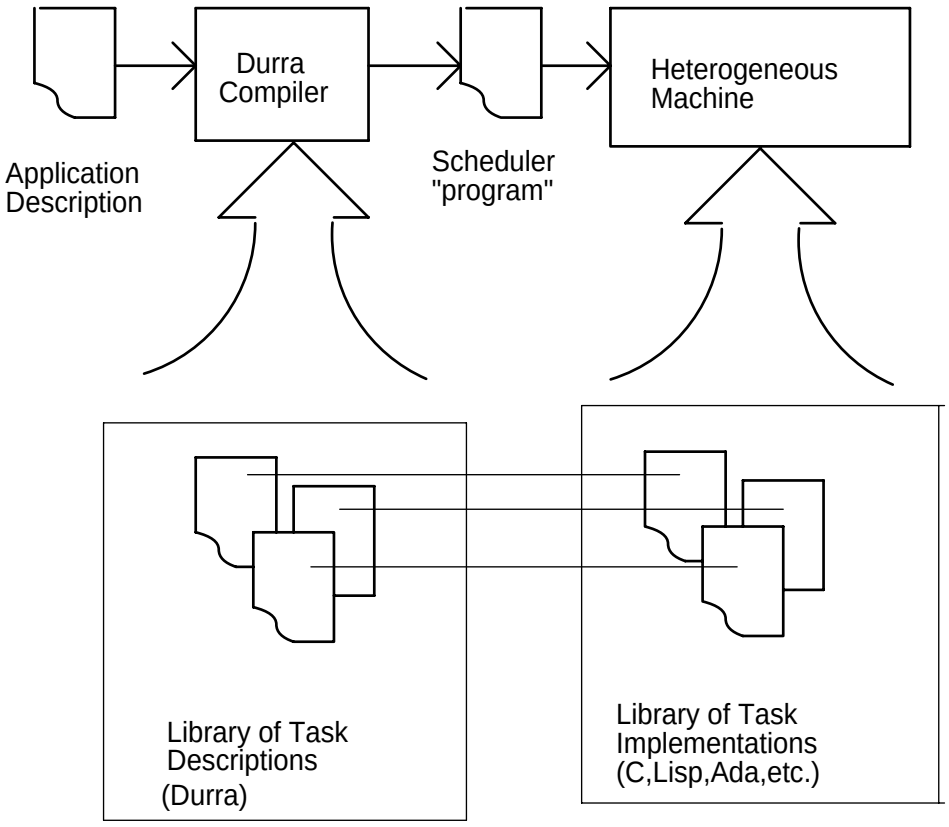


Figure 1-5: Scenario for Developing an Application in Durra

responding to the component tasks) into the processors and issues the appropriate commands to execute the programs.

1.4. Runtime Components

There are three active components in the Durra runtime environment: the application tasks, the Durra server, and the Durra scheduler. Figure 1-6 shows the relationship among these components.

After compiling the type declarations, the component task descriptions, and the application description, as described previously and illustrated in figure 1-5, the application can be executed by performing the following operations:

1. The component task implementations must be stored in the appropriate processors.

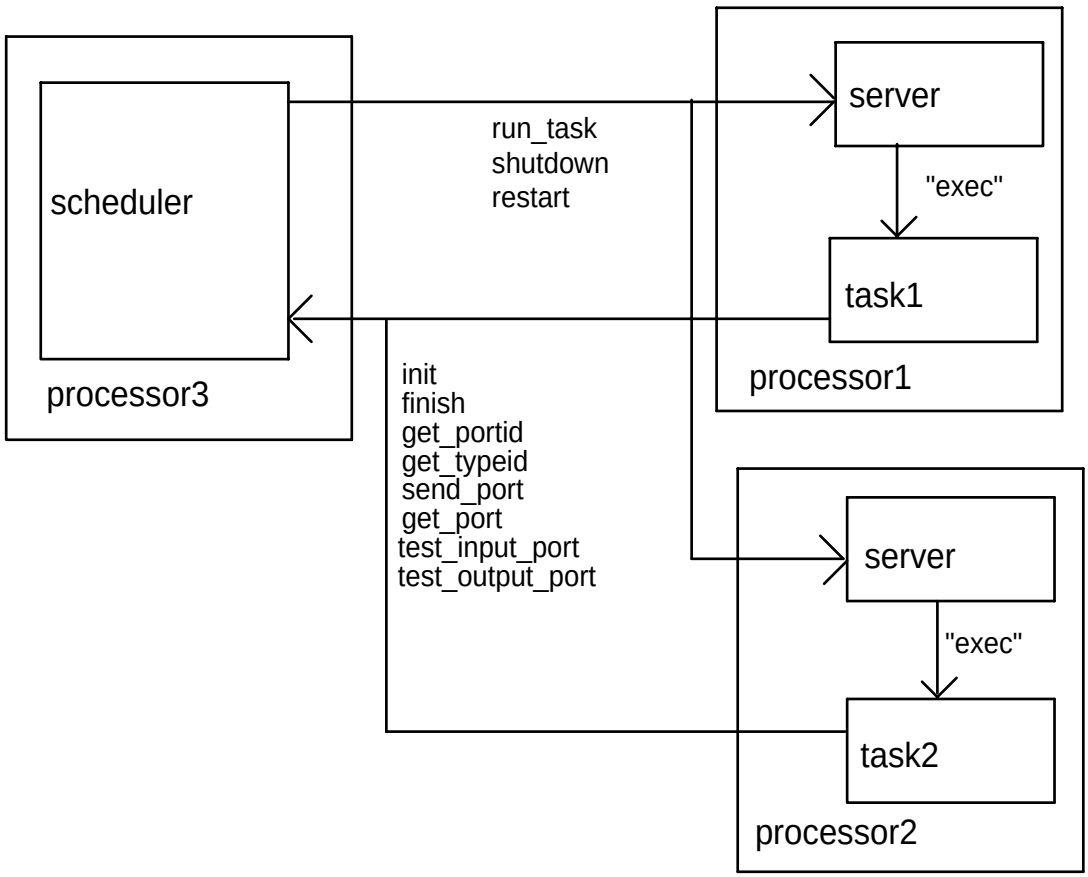


Figure 1-6: The Durra Runtime Environment

2. An instance of the Durra server must be started in each processor.
3. The scheduler must be started in one of the processors. The scheduler receives as an argument the name of the file containing the scheduler program generated by the compilation of the application description. This step initiates the execution of the application.

1.4.1. The Scheduler

The scheduler is the part of the Durra runtime system responsible for starting the tasks, establishing communication links, and monitoring the execution of the application. In addition, the scheduler implements the predefined tasks (**broadcast**, **merge**, and **deal**) and the data transformations described in [1]. The scheduler is invoked with the name of the file containing the scheduler instructions generated by the Durra compiler. A complete description of the scheduler instructions can be found in [3].

After these instructions have been read and processed, the scheduler is ready to start the

execution of the application. In the current UNIX implementation, this is done by performing the following steps:

1. Allocate a UNIX socket for communication with the application tasks.
2. Establish communication with each of the processors running a Durra server.
3. For each of the **task_load** instructions, issue to the appropriate server a **run_task** remote procedure call.
4. Listen in on the UNIX socket allocated in the first step for remote procedure calls from the application tasks.
5. Process the remote procedure calls from the application tasks.

The scheduler waits until all tasks have completed their execution before it, in turn, finishes its execution.

1.4.2. The Server

The server is responsible for starting tasks on its corresponding processor, as directed by the scheduler. One copy of the server must be running on each processor that is to 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 communicates with the server using a set of remote procedure calls to initiate task execution (**run_task**), or to shutdown or restart the server (**shutdown**, and **restart**). Complete details of these remote procedure calls can be found in [3]. The server sits in a loop responding to the requests from the scheduler, executing them as directed.

1.4.3. Application Tasks

The component task implementations making up a Durra application can be written in any language for which a Durra interface has been provided. As of this writing, there are Durra interfaces for both C and Ada. The complete interfaces appear in [3].

When a task is started, the scheduler supplies it with the following information (via a server): 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 in future messages.

Application tasks use the interface to the scheduler 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 following remote procedure calls (RPCs) are provided:

init	Opens a connection to the scheduler.
finish	Informs the scheduler that the task is terminating.
get_portid	Returns a descriptor for the application task to use when referring to a port.

get_typeid	Returns a descriptor for the application task to use when referring to a type.
send_port	Sends data through an output port of the application task.
get_port	Gets data from an input port of the application task.
test_input_port	Tests whether data is available on an input port.
test_output_port	Tests whether there is room in a queue attached to an output port.

Durra tasks typically use these RPCs in the following order:

1. Call **init** to establish communication with the scheduler.
2. Call **get_portid** for each of the task ports (these ports must correspond to the ports used in the task description).
3. Call **get_typeid** for each of the task types (these types must correspond to the data types used in the task description).
4. Call **send_port** and **get_port** as necessary to send and receive data.
5. Call **finish** to break communication with the scheduler.

2. Introduction to the TRW C^3I Testbed

The TRW command, control, communications, and intelligence (C^3I) testbed is a collection of programs that implement a variety of functions found in command, control, communications, and intelligence systems. They have been developed as a part of the Reusable C^3I Node Independent Research and Development Project at TRW Defense Systems Group. The overall objective of this project is to reduce the cost and schedule for fielding state-of-the-art, next generation C^3I systems. TRW plans to achieve this objective by developing a standardized architectural framework for C^3I systems and reusable hardware and software components that are applicable over a wide range of target systems. The technical objectives for the Reusable C^3I Node Project are to 1) develop a reusable C^3I system architecture, including hardware and software components; 2) develop software and, if necessary, hardware components as building blocks within the architecture; and 3) test the flexibility of the architecture and components by demonstrating them on several actual target C^3I systems.

In the remainder of this chapter we will address only one of several tasks identified by TRW as critical to their overall objective, namely the development of an architecture framework in which various C^3I system building blocks fit. This would be used to define the interfaces and the possible combinations of building blocks during the definition of the reusable system baseline, and to help during the integration and testing of the application system. Other major tasks, not addressed in this report, include software component definition and development, hardware component selection and integration, and testbed integration and demonstration.

2.1. C^3I Node Hardware Architecture

The architecture adopted for the reusable C^3I node and its testbed is open so that any vendor's equipment could be connected and the node software integrated with the vendor-supplied hardware and system software. To reach this goal of an open system architecture, it is necessary to have a paradigm for programming loosely-connected networks of multiple special- and general-purpose processors. This is the role played by the Durra language and methodology. The need for experimenting with paradigms for heterogeneous networks provided the motivation for a joint effort between TRW Defense Systems Group and the Software Engineering Institute.

C^3I systems have similar requirements and functions whether the system is for tactical or strategic forces, whether it is fixed-base or mobile, or whether it is controlling planes, tanks, or ships. The purpose of all such systems is to give up-to-the-minute information to commanders at each echelon to assist them in making decisions and to speed implementation of these decisions. The top-level description of a typical tactical C^3I system is shown in Figure 2-1. This system consists of a network of geographically separated nodes where each node is a local area network of processors. Communication among the nodes (located

at command posts) is via combat net radios or land lines and consists of messages whose format and protocol is tightly controlled by Army standards and doctrine. Each node contains a database representing the battle situation (deployment of friendly and enemy forces) and the status of forces controlled by this command post. The overall information flow for the system is predominantly hierarchical, with messages containing orders or requests for information flowing from upper echelons to lower echelons and messages containing status and situation reports flowing back to the upper echelons.

Each node consists of a network of computers whose purpose is to process operator requests for information, to act on incoming messages (by showing them to an operator or replying automatically), and to maintain the situation and status database. The computers at a node may be concentrated in a command post structure (shelter, tent, or parked vehicle) or dispersed among several structures for concealment. They typically communicate via Ethernet or fiber-optic equivalents. At this level the communications protocols are not restricted and may support exchange of message text, database information, keyboard/screen information, etc. At each node in the system common processing is done: communications equipment management, message processing, data management, management of the LAN, and interaction with the operators. In addition, a node may have unique application responsibilities such as resource allocation, planning, and decision making.

2.2. C^3I Node Software Architecture

TRW has defined a nodal software architecture which supports both the common processing functions and site- or system-specific ones. As shown in Figure 2-2, the architecture is extensively layered to support the open architecture concepts described above. In addition to multiple vendors, the architecture supports redistribution of functions among processors to support networks containing from 1 to 50 processors. The reusable C^3I node IR&D Project is building software components for the shaded functional areas in the diagram. These components are implemented as Ada tasks and packages. The tasks communicate via messages using services provided by the Heterogeneous InterProcess Communication (IPC) function; in the testbed, alternative IPC functions have been implemented, one with the Durra runtime environment and one with a TRW-developed IPC. Figure 2-3 shows the top-level data flow in a system constructed from the components.

The ‘‘communication processing’’ area is a support subsystem which performs all of the processing associated with sending and receiving messages to or from other nodes in the system. It typically consists of a collection of hardware and firmware interfaces to the communications media and software to coordinate the operation of the subsystem. The tasks, which are top-level components of the communication processing subsystem, include media service, inbound message processing, outbound message processing, data services (log, archive, etc.), and communication control.

The ‘‘automated message handling (amh)’’ area is a subsystem which performs all process-

Figure 2-1: C^3I System Structure — Army Tactical System

Figure 2-2: Reusable C^3I Node Architecture Layers

Figure 2-3: Data Flow Among Nodal Software Components

ing of the text of incoming and outgoing messages for a node. For incoming messages, it identifies the message type and verifies that it is correctly formatted, extracts data from the message to update the database, and disseminates the message to intended recipients at the workstations. Outgoing message processing consists of review and message release routing within the node. The tasks which are the top-level components of the ‘‘AMH’’ subsystem are inbound message processing, outbound message processing, and control.

The ‘‘system management’’ area performs all of the executive and control functions for the nodal network. This includes:

- system startup, shutdown, and reconfiguration
- user login, access verification, and logout
- error reporting and monitoring
- health and performance monitoring

The components of system management are the overall system manager task and a subsidiary workstation manager task at computers for each operator station. Computers which perform only background processing do not contain a workstation manager.

The ‘‘data management’’ area performs all database access and control functions for a node. In most systems this is a commercial relational database management system such as SYBASE, ORACLE, etc., but may be as simple as a flat-file record handling system which keeps the data in primary memory. The components of this subsystem are the DBMS servers and the distributed server and client control logic.

The ‘‘user/system interface (usi)’’ area manages all interaction with the operators at each workstation in the node. This is a complicated subsystem which contains window managers, main menu (or other user selection mechanism) managers, text and graphic, tool support libraries, and other software to support a multi-window, networked environment.

The ‘‘application support’’ area provides interface and scheduling services for application tasks and contains the common office automation applications like electronic mail, word processing, and spreadsheets. The scheduling services include message event scheduling, database event scheduling, and time scheduling of application tasks.

The definition of the architecture and the implementation of the software components for the C^3I testbed coincide with the Durra paradigm in many ways:

- Utilizing libraries of reusable "black-box" tasks.
- Describing the connectivity of the system outside the code of the task.
- Using message-oriented communication between the tasks.
- Describing data transformations during the transmission of the messages.

The flexibility and control over system operation provided by this paradigm are central to the open architecture concept and the ability to reuse the developed software modules. In the

remainder of this report, we will illustrate the use of Durra to build applications consisting of C^3I node software modules in which the various tasks connected through queues exchange messages.

3. Task and Application Descriptions

In this chapter we illustrate the various kinds of descriptions that would be written in a typical application of Durra. The examples depict increasingly more complex task descriptions. We first show a simple *task description*, of a task implemented by a single program and capable of running on one of the processors in a heterogeneous network. We then show a *compound task description*, that is, a task that consists of several simpler tasks whose ports are connected via data queues. Finally, we show an *application description* which, syntactically, looks like a compound task description and ties together all of the component tasks (simple or otherwise) of an application.

3.1. A Simple Task Description and Its Implementation

In this section we illustrate the relation between a Durra task description, a task implementation, and the interface between the task implementation and the Durra runtime environment. The task we have selected is the control module of the Automatic Message Handler in the C^3I node.

As described in Section 2.2, the automated message handling subsystem of the C^3I node performs all processing of the text of incoming and outgoing messages for a node. As we show in this and in later sections, this subsystem consists of several tasks. One of these tasks, ‘‘amhs_control,’’ is responsible for the overall operation of the subsystem and performs its job by sending requests to the other components of the subsystem and processing their responses. In addition, it provides the interface to other subsystems of the C^3I node, specifically, the system manager subsystem.

```
task amhs_control
ports
        SM_In   : in system_command;
        SM_Out  : out subsystem_response;
        Cmd_Out : out system_command;
        Resp_In : in subsystem_response;
attributes
        implementation = "amh_control";
        x10window = "=80x12+0+0";
        x11window = "-geometry 80x12+0+0";
        processor = sun;
end amhs_control;
```

Figure 3-1: AMHS_Control Task Description

Figure 3-1 shows the Durra description of the ‘‘amhs_control’’ task. The interface portion of the task description indicates that it consumes data of type ‘‘system_command’’ and ‘‘subsystem_response’’ through input ports ‘‘SM_In’’ and ‘‘Resp_In,’’ and produces data of type ‘‘subsystem_response’’ and ‘‘system_command’’ through output ports ‘‘SM_Out’’ and ‘‘Cmd_Out,’’ respectively.

The attributes of the task description provide additional information about the task. These include the name of the task implementation (i.e., ‘‘amh_control’’ is the actual executable program), the name of the processor or processor class on which this implementation can execute (‘‘sun’’), and the specifications of a window geometry to be used for communication between a human operator and the task. Notice that two alternative window specifications are provided, depending on which version of the window manager is running on the processor to which the task will be allocated at runtime.

As indicated in Figure 1-5, for each task description, there is a corresponding task implementation. The latter is a program that will be loaded, at runtime, on one of the machines on the network and will communicate with other task implementations through ports. The task description indicates that this is a program that can run on a Sun workstation and that it communicates with a human operator through a small window (12 lines of 80 characters). This window specification is necessary because this task runs as an independent, interactive program, with its own terminal emulator running in the window. The server [3] running on the Sun workstation uses the window specification as part of the UNIX task initiation command.

```
with Ipc_Types;         use Ipc_Types;
with Text_Io;           use Text_Io;
with Interface;

procedure AMH_Control is
    pragma Priority(4);
    Process_ID                : Integer := 1001;
    size                      : Integer;
    bound                     : Integer;
    Verbose                   : constant Boolean := True;

    SM_In_Port          : Positive;
    SM_Out_Port         : Positive;
    Cmd_Out_Port        : Positive;
    Resp_In_Port        : Positive;

    procedure AMH_Control_Processing( Process_ID : in Integer)
        is separate;

begin
        Put_Line( "***** amh_control *****" );
        Interface.Init;
        Interface.Get_PortID("SM_In", SM_In_Port, bound, size);
        Interface.Get_PortID("SM_Out", SM_Out_Port, bound, size);
        Interface.Get_PortID("Cmd_Out", Cmd_Out_Port, bound, size);
        Interface.Get_PortID("Resp_In", Resp_In_Port, bound, size);

        Amh_Control_Processing (Process_ID);

        Interface.Finish;
        return;

end AMH_Control;
```

Figure 3-2: AMHS_Control Task Implementation

The task implementation in this example is an Ada program, as shown in figure 3-2. This program consists of a main unit, the procedure listed in the Figure (‘‘AMH_Control’’), and a separately compiled procedure (‘‘AMH_Control_Processing’’), not shown here. The main unit in this program is responsible for 1) establishing a connection with the Durra scheduler (call to ‘‘Init’’), 2) getting a small integer token to identify each of its ports (calls to ‘‘Get_PortID’’). 3) invoking the separate procedure, where the bulk of the application specific work is done, and, finally, 4) notifying the Durra scheduler when it has completed its execution (call to ‘‘Finish’’).

Notice the correspondence between the port names used in the calls to ‘‘Get_PortID’’ in the task implementation of Figure 3-2 and the port names used in port declarations in the task description of Figure 3-1.

The separately compiled procedure (‘‘AMHS_Control_Implementation’’) uses the port identifier tokens collected by the main procedure to invoke various input and output operations on these ports. This procedure is rather lengthy, and instead of including the entire text , we illustrate these port operations via a small segment of ‘‘AMH_Control_Processing’’, the procedure (‘‘Read_Next_Message’’) shown in Figure 3-3. The small procedure loops continuously, testing for the presence of messages on the input ports (calls to ‘‘Test_Input_Port’’). If there is at least one message on either of the ports, the procedure reads the message into a buffer (calls to ‘‘Get_Port’’) and returns to the caller.

There are a few more interface procedures to support the communication between the user task implementations and the Durra scheduler. However, for the purposes of this report, it is not necessary to go into further details. The complete interface to the scheduler is described in [3]. Appendix G contains the specifications of the interface package for Ada programs.

3.2. A Compound Task Description and Its Structure

The Durra language supports hierarchical task descriptions. The previous section illustrated a simple task description and its implementation. The task used in that example is one of several components of the Automated Message Handling System (AMHS). In this section we will use this and other similar, simple tasks to describe the complete message handler shown in figure 3-4.

The ‘‘AMHS’’ subsystem consists of five tasks. Three of these tasks, ‘‘AMHS_control,’’ ‘‘AMHS_inbound,’’ ‘‘AMHS_outbound,’’ are user-implemented (‘‘AMHS_control’’ was shown in the previous section, the other two appear in Appendix B). The other two tasks, ‘‘broadcast’’ and ‘‘merge’’ are predefined in the Durra language and implemented directly by the Durra runtime system. 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 (specified in the task selection) and copies them into a single output port.

```
procedure Read_Next_Message(Msg_Type: in out Message_Type) is
    Read_Cycle_Delay : Duration := 0.5;
    Durra_type, size, n    : natural;
begin
    Read_Loop:
        loop
            Interface.Test_Input_Port( SM_In_Port,
                                       Durra_Type,
                                       size,
                                       n);
            if n > 0 then
                Interface.Get_Port( SM_In_Port,
                                    Cmd_Msg'Address,
                                    size,
                                    Msg_Type);
                exit Read_Loop;
            end if;
            Interface.Test_Input_Port( Resp_In_Port,
                                       Durra_type,
                                       size,
                                       n);
            if n > 0 then
                Interface.Get_Port( Resp_In_Port,
                                    Rsp_Msg'address,
                                    size,
                                    Msg_Type);
                exit Read_Loop;
            end if;
            delay Read_Cycle_Delay;
        end loop Read_Loop;
    return;
end Read_Next_Message;
```

Figure 3-3: Sample Sequence of Port Operations

The application description instantiates five processes (‘‘ac,’’ ‘‘ai,’’ ‘‘ao,’’ ‘‘pb,’’ and ‘‘pm’’), corresponding to the five tasks mentioned above and six queues (‘‘q1’’ through ‘‘q6’’) that connect the processes’ ports. Notice that not all the internal-task ports are connected. This group of tasks implements a subsystem which can be used as a building block for a larger system (the C^3I node, in this case,) and therefore must also have ports to communicate with other subsystems. Since the subsystem is really an abstraction and does not really correspond to a executable program, its ports (‘‘SM_Commands,’’ ‘‘SM_Response,’’ ‘‘COMM_Inbound,’’ ‘‘COMM_Outbound,’’ ‘‘WS_Inbound,’’ and ‘‘WS_Outbound’’) must be implemented by internal-task ports. This is the purpose of the **bind** declarations, which declare the internal-task ports that implement or correspond to the subsystem ports.

3.3. An Application Description

In this section we illustrate a complete *application description*. There are no special language features beyond those used to describe a compound task. 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.

```
task AMHS
-- Automated Message Handling Subsystem
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 3-4: AMHS Subsystem Description

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 scheduler by executing an optional *code generation phase* of the Durra compiler, as described in [3].

Continuing with the examples of the previous sections, let's assume that the C^3I node constitutes the complete application (i.e., we are ignoring the rest of the network -- it would consist of multiple instances of nodes and communication tasks). In addition to the Automated Message Handling System, there are several other components of the C^3I node, as illustrated in figure 3-5.

```
task configuration
structure
    process
      --    real system processes
        sm      : task system_manager;
        com     : task comm;
        amh     : task amhs;
        w1p     : task wkstn;
        w2p     : task wkstn;
      --    auxiliary system processes
        dmxp: task broadcast    -- message demultiplexor
                ports
                        in1        : in  workstation_if_message;
                        out1, out2 : out workstation_if_message;
              end broadcast;
        muxp: task merge        -- message multiplexor
                ports
                        out1     : out comm_if_message;
                        in1, in2 : in  comm_if_message;
                attribute
                        mode = fifo;
              end merge;
        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    >> dmxp.in1;
        q_i3    : dmxp.out1         >> w1p.Inbound;
        q_i4    : dmxp.out2         >> w2p.Inbound;
        -- outbound message propagation
        q_o1    : w1p.Outbound      >> muxp.in1;
        q_o2    : w2p.Outbound      >> muxp.in2;
        q_03    : muxp.out1         >> amh.WS_Outbound;
        q_o4    : amh.COMM_Outbound >> com.Outbound;
end configuration;
```

Figure 3-5: C^3I Node Description

The C^3I node consists of five main subsystems (‘‘system_manager,’’ ‘‘comm,’’ ‘‘amhs,’’ and two instances of ‘‘wkstn’’). In addition, there are four predefined tasks (two broadcast and two merge) that serve to multiplex and demultiplex messages exchanged between the main subsystems, as shown in Figure 3-6.

The complete set of task descriptions and type declarations used to build the application are included in the appendices.

Figure 3-6: C^3I Node Structure

4. Software Development Methodologies

A great deal of effort has been devoted to the development of improved software development process models. As described in [5], models have evolved from the early ‘‘code-and-fix’’ model, through the ‘‘stagewise’’ and ‘‘waterfall’’ models (which attempt to bring order to the process by recognizing formal steps in the process), through the ‘‘evolutionary’’ and ‘‘transform’’ models (which attempt to address the need for experimentation, refinement of requirements, and automation of the code generation phase.) The spiral model of the software process has evolved over several years at TRW, based on experience on a number of large software projects and, as indicated in [5], accommodates most previous models as special cases.

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

One of the advantages of this model is that it allows the identification of areas of uncertainty that are significant sources of risk. Once these critical areas are identified, the spiral model allows for the selective application of an appropriate development strategy to these risk areas first. Thus, while at first sight the spiral model looks no better than the waterfall model, a key difference is that the spiral allows the designers to concentrate on selected problem areas rather than following a predetermined order. Once the highest-risk problem has been taken care of, the next higher risk area can be attacked, and so on.

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

4.1. Durra as a Tool for Successive Refinement

The programming paradigm embodied in Durra fits very naturally this style of software development. Although we don’t claim to have solved all problems or identified all the necessary tools, we would like to suggest that a language like Durra would be of great value in the context of the spiral model. It would allow the designer to build mock-ups of an application, starting with a gross decomposition into tasks described by templates that are specified by their interface and behavioral properties. Once this is completed, the application can be emulated using tools like MasterTask [4] 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

specifications are more critical or demanding. In other words, the purpose of this initial emulation is to identify the component task more likely to affect the performance of the entire system. The designers can then experiment by writing alternative behavioral specifications for the offending task until a satisfactory specification (i.e., template) is obtained. Once this is achieved, the designers can proceed by replacing the original task descriptions with more detailed templates, consisting of internal tasks and queues, using the structure description features of Durra. These more refined application descriptions can again be emulated, experimenting with alternative behavioral specifications of the internal tasks, until a satisfactory internal structure (i.e., decomposition) has been achieved. This process can be repeated as often as necessary, varying the degree of refinement of the tasks, and even backtracking if a dead-end is reached. It is not necessary to start coding a task until later, when its specifications are acceptable, and when the designers decide that it should not be further decomposed.

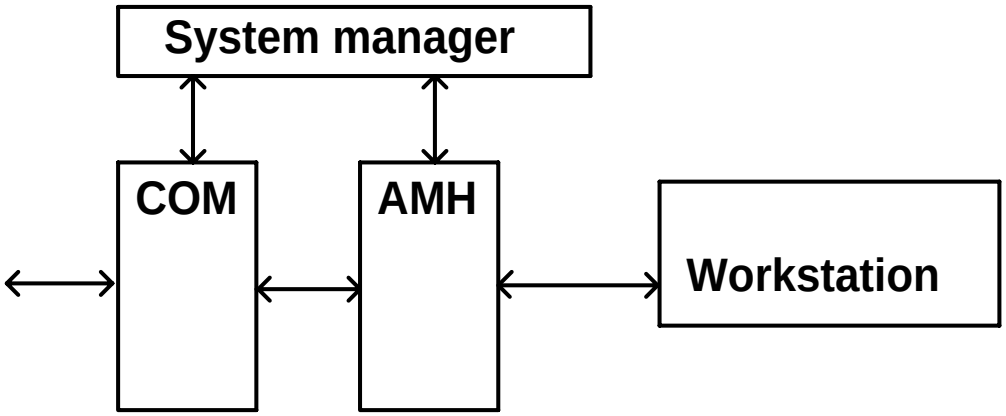
Of course, it is quite possible that a satisfactory specification might be impossible to meet and a task implementation might have to be rejected. The designers would then have to backtrack to an earlier, less detailed design and try alternative specifications, or even alternative decompositions of a 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.

4.2. Incremental Development of the C^3I Node

To illustrate an incremental development process using Durra, in this section we will show three alternative development sequences for the C^3I node. In all three cases we start from the same basic configuration for the node shown in Figure 4-1. The node consists of four subsystems: System Manager, Communications, Application Message Handler, and Workstation.

These subsystems correspond to the first four process declarations in the structure of the Durra application description in Figure 4-1. In addition, the application description declares two auxiliary tasks which are used for communications between the system manager and the communication and application message handler subsystems. These auxiliary tasks broadcast commands from the system manager to the other two subsystems, and merge their responses to the system manager. The queue declarations in the Figure connect the task ports in the appropriate manner.

For brevity, we are not showing a simpler configuration that could have been our initial design. For example, we could have started with a node consisting of two subsystems: (combined) Communications, and Workstation. A first step could be to divide the first subsystem into three components: System Manager, Application Message Handler, and Communications proper. We take this as our initial design.




```
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 4-1: Initial Application Description

Figure 4-2 shows the tasks descriptions for the subsystems of the initial configuration. Since at this stage of the development we might not be ready to commit to any particular structure for the subsystems, these are described as simple, unstructured tasks. This information is sufficient to do static checks, including port (i.e., type) compatibility and graph connectivity. We could continue the design in this fashion, successively refining the subsystem descriptions until, at the end, the application is fully described as a hierarchical graph structure in which the innermost nodes are implemented as separate programs, specified by the ‘‘implementation’’ attribute of the corresponding task description.

However, if we want to carry out some preliminary dynamic tests, 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’’ [4] as a subsystem emulator. To use MasterTask, we need to include a timing expression in the subsystem task description, and specify ‘‘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 shown in Figure 4-2.

Figures 4-4, 4-5, and 4-6 show three possible design sequences starting from the original design of Figure 4-1 and ending with the final design shown in Figure 4-3. In Figure 4-4 we first replicate the workstation, then we decompose the communication and application message handlers, and finally we decompose the workstations. In Figure 4-5 we first decompose the communication and application message handler subsystems, then we replicate the workstation, and finally we decompose the workstation. In Figure 4-6 we carry out the design in yet a different order. First we decompose the communication and application message handler subsystems, then we decompose the workstation, and finally we replicate the workstation.

There are a number of other alternative design sequences leading to the same final design and we do not need to belabor the point.

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

a -- System Manager Subsystem

```
task comm
   -- red communications processing
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;
```

b -- Communications Subsystem

```
task AMHS
-- Automated Message Handling Subsystem
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;
```

c -- Application Message Handler Subsystem

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

d -- Workstation Subsystem

Figure 4-2: Initial Task Descriptions

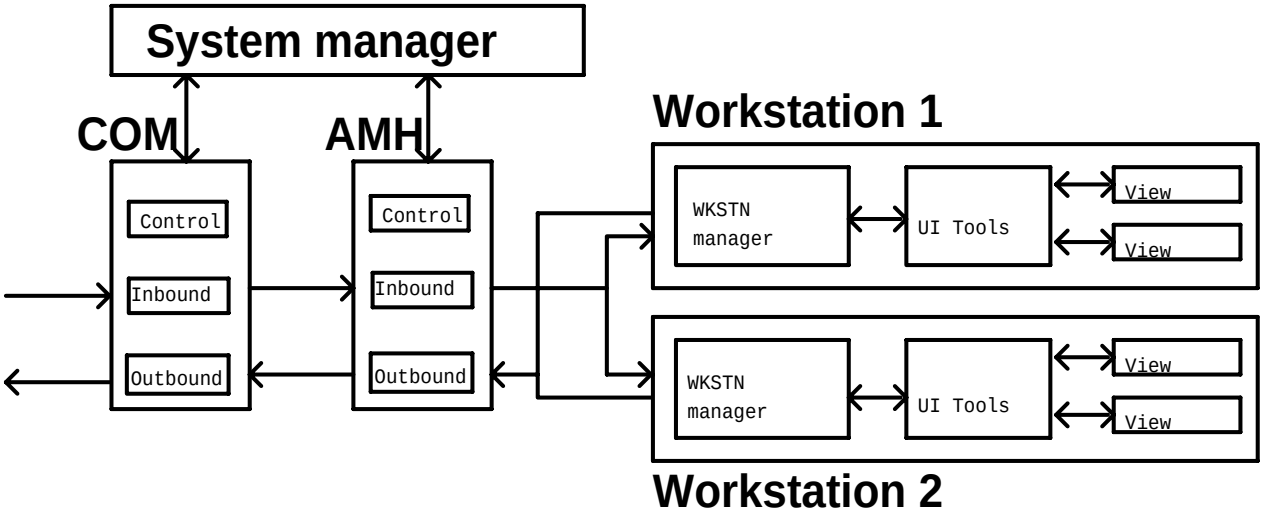


Figure 4-3: Final Structure of the C^3I Node

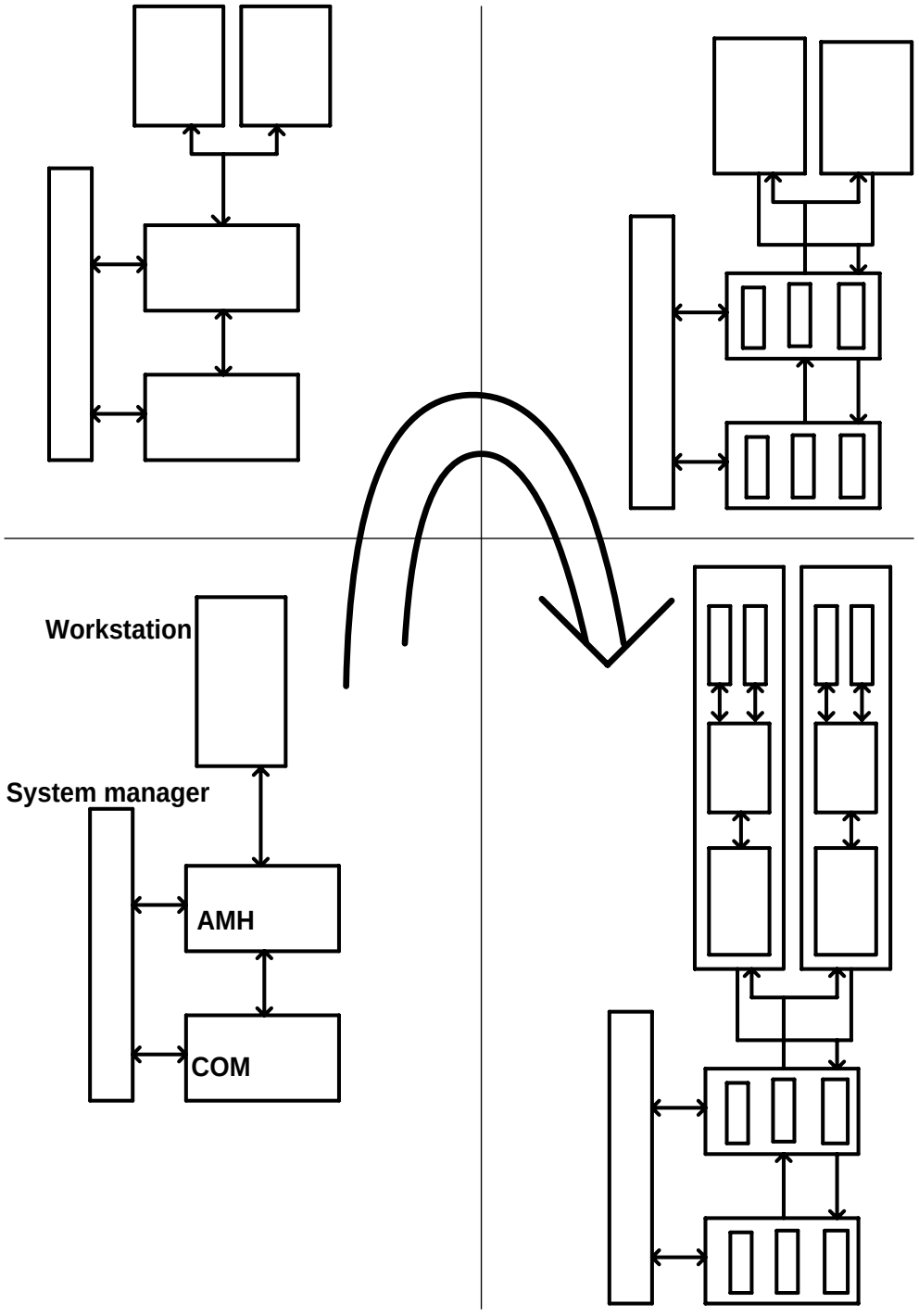


Figure 4-4: Incremental Development of the C^3I Node — Sequence 1

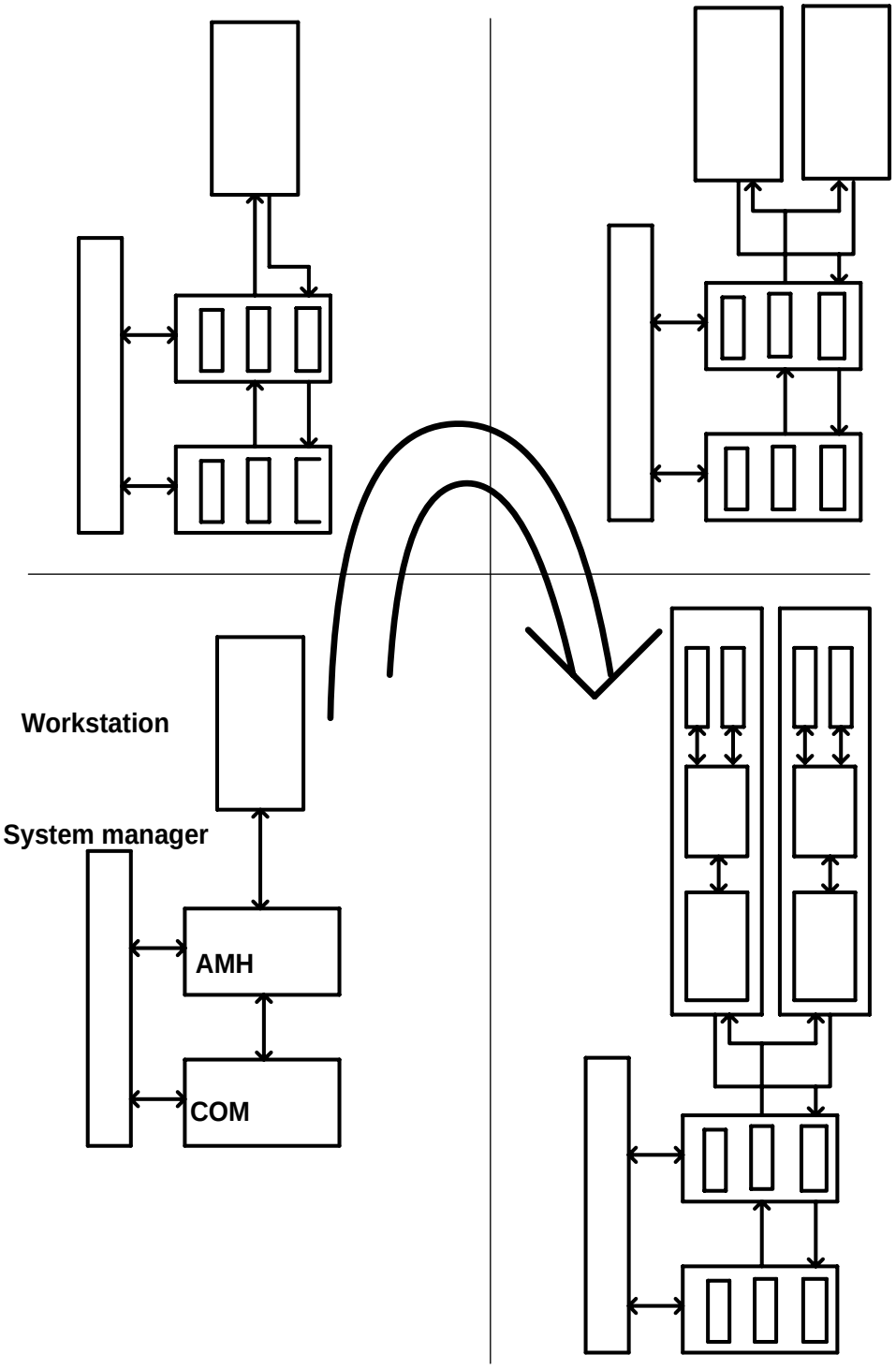


Figure 4-5: Incremental Development of the C^3I Node — Sequence 2

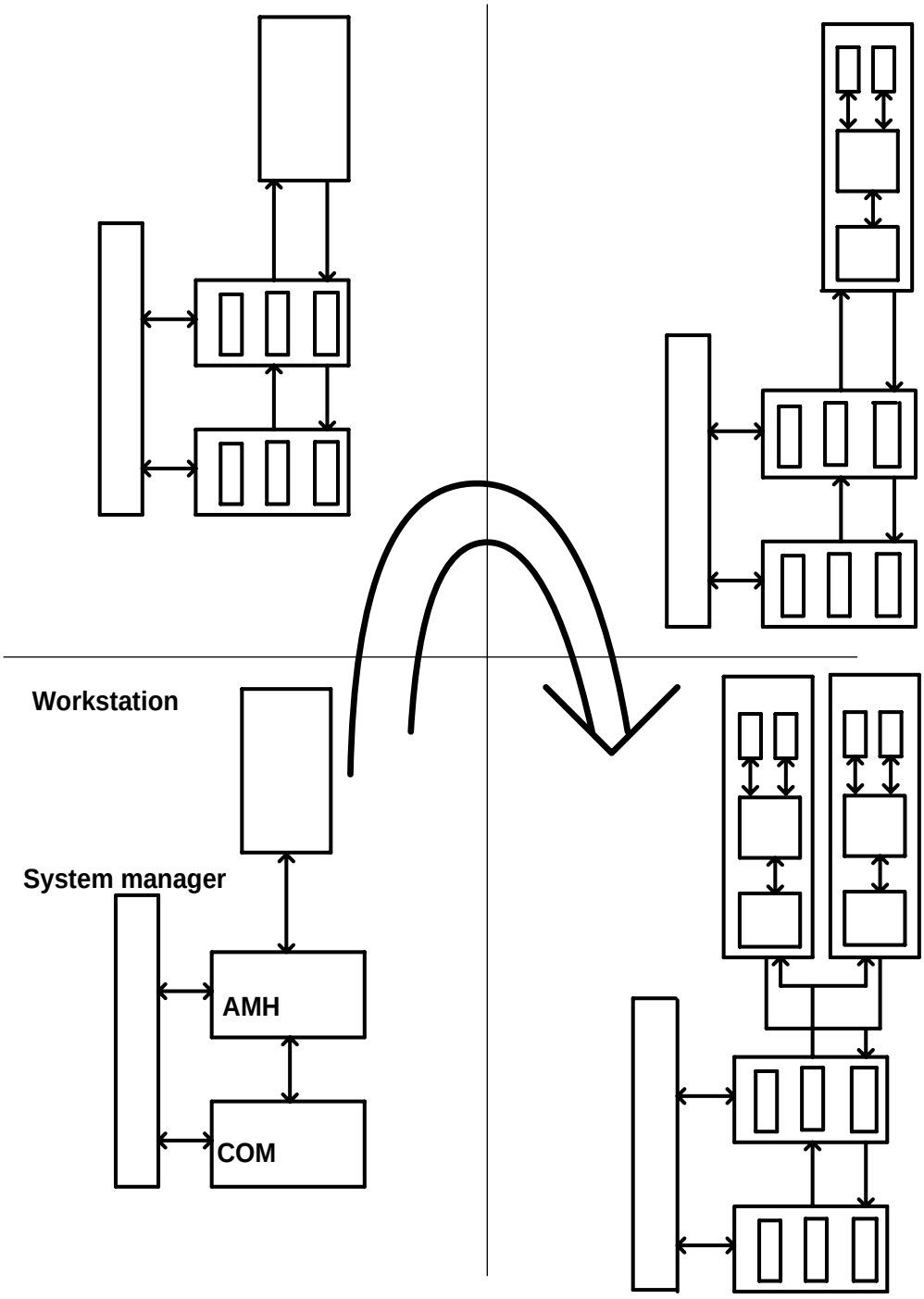


Figure 4-6: Incremental Development of the C^{3}I Node — Sequence 3

5. Conclusions and Future Developments

This report contains the results of the first phase of a joint project between the Software Engineering Institute and TRW Defense Systems Group. The objective of this project is to demonstrate a new technology for developing distributed applications on networks of heterogeneous machines. We have chosen as the initial application domain prototype C^3I applications built using the C^3I testbed developed by TRW and the Durra language and methodology developed by the SEI.

To help provide direction to the effort and to evaluate our progress we identified three major phases: Basic Demonstration, Study of Networking Issues, and Advanced Demonstration. This report describes the results of the first phase. The second and third phases will be the subject of a report scheduled for the fall of 1989.

5.1. Basic Demonstration

The goal of the first phase was to demonstrate the basic functionality of a command and control application developed using Durra and the capability of supporting heterogeneous networking. In addition to demonstrating the application development methodology, the sample applications developed for the demonstration will provide an experimental facility for future performance measurements and evaluations. We accomplished this goal through several intermediate subgoals:

Defined common communication/scheduler primitives. The Durra model of computation assumes the use of a small set of message passing primitive operations. The TRW testbed tasks also use a message passing paradigm but their original interface was not as narrowly defined as in Durra. The narrow interface was adopted as the first step of the project and we modified the testbed software to use the Durra scheduler interface.

Developed Durra descriptions of C^3I tasks, subsystems, and node. We analyzed the testbed modules and developed Durra descriptions of the tasks. Writing these descriptions required an understanding of the behavior of the testbed tasks, their performance and communication requirements (ports and message types), and the structure of typical applications in the C^3I domain. We also developed Durra descriptions of the subsystems, defined by TRW as the architecture of the C^3I node.

Demonstrated the methodology. All of the Durra task descriptions and the modified testbed modules were compiled and the distributed C^3I node executed as expected, on the computer networks at the SEI and at TRW Defense Systems Group, using two types of workstations (MicroVax and Sun), two versions of UNIX (Ultrix-32 and BSD-4.3), and two versions of X windows (X10 and X11).

5.2. Study of Advanced Networking Issues

Following the successful demonstration of the basic methodology, we need to explore the extension or modification of the Durra paradigm of networking to meet more realistic C^3I requirements.

System robustness issues. The current implementation of Durra uses a centralized scheduler. This was done mostly for expediency, to get the system up and running in a short time. It was always understood that a centralized critical resource was not acceptable as a long term solution. We are in the process of designing and implementing a distributed scheduler to enhance the reliability and availability of the applications.

Dynamic application startup/reconfiguration. The initial reconfiguration features of Durra are based on a preset (i.e., compile time) description of the conditions for a change and the nature of the change in the configuration of an application. This model might be too restrictive for a number of applications, and we will investigate the issues involved in developing an *interactive scheduler interface* to allow human operators to invoke arbitrary application reconfigurations.

Default reconfiguration actions. These two styles of reconfiguration (preset and interactive) might not be adequate to cope with all possible system anomalies that would trigger a reconfiguration. A Durra description that would attempt to cover all possible anomalies will be unwieldy, unreadable, and most likely incomplete. By the same token, an operator might be swamped by the speed of events or confused as to the best action to take under real-time conditions. Thus, there is need for default reconfiguration actions to be followed by the scheduler. These defaults should not be built-in but rather should be a property of the application and might vary over time, during the life of the application. We will investigate appropriate policies for taking default reconfiguration actions and the languages for specifying these policies.

5.3. Advanced Demonstration

The goal of this phase will be to demonstrate dynamic reconfiguration features (preset and interactive) to support fault tolerance. This is planned to be completed by September 1989. To achieve this goal, we have adopted the following intermediate subgoals:

Demonstrate preset dynamic reconfiguration. Augment the Durra application descriptions developed during the first phase (these descriptions are contained in this report) with dynamic reconfiguration statements. This requires the identification of typical reconfiguration requirements in C^3I applications and their expression in Durra.

Develop/demonstrate interactive dynamic reconfiguration. We will implement a *system operator* program to allow direct communication between an operator and the runtime scheduler. This program will consist primarily of a simple user interface and a command interpreter that will take requests from an operator and will translate these into scheduler instructions to perform interactive reconfigurations.

Develop/demonstrate policy language concepts. We will implement a simple policy language and associated mechanisms to handle system anomalies when neither the preset nor interactive reconfiguration facilities are adequate.

References

[1] M.R. Barbacci and J.M. Wing.
Durra: A Task-Level Description Language.
Technical Report CMU/SEI-86-TR-3, DTIC: ADA178975, Software Engineering Institute, Carnegie Mellon University, December, 1986.

[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, D.L. Doubleday, and C.B. Weinstock.
The Durra Runtime Environment.
Technical Report CMU/SEI-88-TR-18, DTIC: ADA199480, Software Engineering Institute, Carnegie Mellon University, July, 1988.

[4] M.R. Barbacci.
MasterTask: The Durra Task Emulator.
Technical Report CMU/SEI-88-TR-20, DTIC: ADA199429, Software Engineering Institute, Carnegie Mellon University, July, 1988.

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

Appendix A: C^{3}I Node Application Description

```
task configuration
structure
    process
      --    real system processes
        sm      : task system_manager;
        com     : task comm;
        amh     : task amhs;
        w1p     : task wkstn;
        w2p     : task wkstn;
      --    auxiliary system processes
        dmxp: task broadcast    -- message demultiplexor
                ports
                        in1        : in  workstation_if_message;
                        out1, out2 : out workstation_if_message;
              end broadcast;
        muxp: task merge        -- message multiplexor
                ports
                        out1     : out comm_if_message;
                        in1, in2 : in  comm_if_message;
                attribute
                        mode = fifo;
              end merge;
        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    >> dmxp.in1;
        q_i3    : dmxp.out1         >> w1p.Inbound;
        q_i4    : dmxp.out2         >> w2p.Inbound;
        -- outbound message propagation
        q_o1    : w1p.Outbound      >> muxp.in1;
        q_o2    : w2p.Outbound      >> muxp.in2;
        q_03    : muxp.out1         >> amh.WS_Outbound;
        q_o4    : amh.COMM_Outbound >> com.Outbound;
end configuration;
```

Appendix B: Application Message Handler Description

B.1. Subsystem Description

```
task AMHS
-- Automated Message Handling Subsystem
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;
```

B.2. Component Task Descriptions

B.2.1. Control Task

```
task amhs_control
ports
        SM_In   : in system_command;
        SM_Out  : out subsystem_response;
        Cmd_Out : out system_command;
        Resp_In : in subsystem_response;
attributes
        implementation = "amh_control";
        x10window = "=80x12+0+0";
        x11window = "-geometry 80x12+0+0";
        processor = sun;
end amhs_control;
```

B.2.2. Inbound Message Task

```
task amhs_inbound
ports
        Cmd_In          : in  system_command;
        Resp_Out        : out subsystem_response;
        COMM_Inbound    : in  comm_if_message;
        WS_Inbound      : out workstation_if_message;
attributes
        implementation = "amh_inbound";
        x10window = "=80x12+0+190";
        x11window = "-geometry 80x12+0+190";
        processor = sun;
end amhs_inbound;
```

B.2.3. Outbound Message Task

```
task amhs_outbound
ports
        Cmd_In          : in  system_command;
        Resp_Out        : out subsystem_response;
        COMM_Outbound   : out comm_if_message;
        WS_Outbound     : in  comm_if_message;
attributes
        implementation = "amh_outbound";
        x10window = "=80x12+0+380";
        x11window = "-geometry 80x12+0+380";
        processor = sun;
end amhs_outbound;
```

Appendix C: Communications

C.1. Subsystem Description

```
task comm
   -- red communications processing
ports
        SM_Commands     : in  system_command;
        SM_Responses    : out subsystem_response;
        Inbound         : out comm_if_message;
        Outbound        : in  comm_if_message;
structure
        process
                cc      : task comm_control;
                ci      : task comm_inbound;
                co      : task comm_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     = cc.SM_In;
                SM_Responses    = cc.SM_Out;
                Inbound         = ci.Inbound;
                Outbound        = co.Outbound;
        queue
                q1      : cc.Cmd_Out    >> pb.in1;
                q2      : pb.out1       >> ci.Cmd_In;
                q3      : pb.out2       >> co.Cmd_In;
                q4      : ci.Resp_Out   >> pm.in1;
                q5      : co.Resp_Out   >> pm.in2;
                q6      : pm.out1       >> cc.Resp_In;
                q7      : co.Echo_Out   >> ci.Echo_In;
end comm;
```

C.2. Component Task Descriptions

C.2.1. Control Task

```
task comm_control
ports
        SM_In   : in system_command;
        SM_Out  : out subsystem_response;
        Cmd_Out : out system_command;
        Resp_In : in subsystem_response;
attributes
        implementation = "rcom_control";
        x10window = "=80x12+510+0";
        x11window = "-geometry 80x12+510+0";
        processor = sun;
end comm_control;
```

C.2.2. Inbound Message Task

```
task comm_inbound
ports
        Cmd_In          : in  system_command;
        Resp_Out        : out subsystem_response;
        Inbound         : out comm_if_message;
        Echo_In         : in  comm_if_message;
attributes
        implementation = "rcom_inbound";
        x10window = "=80x12+510+190";
        x11window = "-geometry 80x12+510+190";
        processor = sun;
end comm_inbound;
```

C.2.3. Outbound Message Task

```
task comm_outbound
ports
        Cmd_In          : in  system_command;
        Resp_Out        : out subsystem_response;
        Outbound        : in  comm_if_message;
        Echo_Out        : out comm_if_message;
attributes
        implementation = "rcom_outbound";
        x10window = "=80x12+510+380";
        x11window = "-geometry 80x12+510+380";
        processor = sun;
end comm_outbound;
```

Appendix D: System Manager Description

```
task system_manager
ports
        SM_In   : in  subsystem_response;
        SM_Out  : out system_command;
attributes
        implementation = "system_manager";
        x10window = "=80x12+200+570";
        x11window = "-geometry 80x12+200+570";
        processor = sun;
end system_manager;
```

Appendix E: Workstation Description

To preserve proprietary information, we do not show the various tasks constituting the user's workstation subsystem. For the purposes of our demonstration, their functions were carried out by the workstation manager task.

E.1. Subsystem Description

```
task wkstn
ports
        Inbound         : in  workstation_if_message;
        Outbound        : out comm_if_message;
structure
    process
        wm: task wkstn_manager;
    bind
        Inbound         = wm.Inbound;
        Outbound        = wm.Outbound;
end wkstn;
```

E.2. Component Task Descriptions

E.2.1. Manager Task

```
task wkstn_manager
ports
        Inbound         : in  workstation_if_message;
        Outbound        : out comm_if_message;
attributes
        processor = sun;
        x10window = "=80x12+0+740";
        x11window = "-geometry 80x12+0+740";
        implementation = "usi_sim";
end wkstn_manager;
```

Appendix F: Type Declarations

```
type byte is size 8;
type comm_if_message is array of byte;
type string is array of byte;
type subsystem_response is array of byte;
type system_command is array of byte;
type workstation_if_message is array of byte;
```

Appendix G: Scheduler Interface for Ada Task Implementations

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

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

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

  procedure Finish;

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

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

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

  procedure Init;

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

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

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

end Interface;
```

Table of Contents

List of Figures