Technical Report

CMU/SEI-91-TR-22
ESD-91-TR-22

# Building Distributed Ada Applications from Specifications and Functional Components

Dennis L. Doubleday
Mario R. Barbacci
Charles B. Weinstock
Michael J. Gardner
Randall W. Lichota

December 1991

Technical Report
CMU/SEI-91-TR-22
ESD-91-TR-22
December 1991

# Building Distributed Ada Applications from Specifications and Functional Components

Dennis L. Doubleday
Mario R. Barbacci
Charles B. Weinstock
Michael J. Gardner
Randall W. Lichota

Distributed Systems Project

Software Engineering Institute
Carnegie Mellon University
Pittsburgh, Pennsylvania 15213

# Table of Contents

# List of Figures

# Building Distributed Ada Applications from Specifications and Functional Components

**Abstract:** Durra is a language and support environment for the specification and execution of distributed Ada applications. A Durra programmer describes an application as a collection of processes and data links. More complicated application descriptions may also include a structuring of this collection that varies dynamically according to a set of reconfiguration conditions. Each process defined in the application description is associated with an independently compiled Ada subprogram that implements the behavior of that process. The Durra programmer specifies the distribution of application components by assigning them to virtual nodes called clusters. For each cluster, the Durra compiler generates a multithreaded Ada program that imports the code for the processes assigned to that node and manages their execution. Durra also facilitates rapid prototyping through the use of tools that interpret timing specifications associated with processes and generate Ada code to simulate their expected behavior.

## 1 Motivation

Computing environments consisting of loosely connected networks of special and general purpose processors are becoming commonplace. The corresponding trend in software is away from sequential programs running on large uniprocessor hardware toward concurrent programs distributed across multiple, possibly heterogeneous, platforms. Today, developers of such applications typically “hard code” the allocation of computing resources into their applications by explicitly assigning specific tasks to run on specific processors at specific times. The component tasks of such an application require built-in knowledge about the structure of the application and the allocation of resources in order to communicate with other tasks. This coupling of function to structure complicates modification of the application, poses obstacles to runtime changes in the application structure, and prevents reuse of the tasks in other environments. Developers new tools that allow them to abstract application structure from function.

In this paper we describe Durra, a language and support environment for the specification and execution of distributed Ada applications[1].

[1]. An earlier version of this paper was presented at TRI-Ada’91, San Jose, California, October 22-25, 1991, and appears in the conference proceedings.

# 2 Introduction to Durra

## 2.1 The Durra Language

Durra [1] is a task-level application description language.[2] The basic building blocks of the language are the *task description*, which specifies the properties of an associated subprogram or subsystem, and the *channel description*, which specifies the properties of an Ada package implementing a communication facility. See Figure 2-1 for a Durra task description template. Task descriptions may be either *primitive* or *compound*. A primitive task description represents a single thread of control.[3] A compound task description is a composition of other task and channel descriptions. Channel descriptions are syntactically similar to primitive task descriptions although the implementations exhibit different behaviors. Task implementations are active components; they initiate requests to send or receive messages by calling procedures provided by the runtime environment. Channel implementations are passive components; they wait for and respond to requests from the runtime environment. Task and channel implementations are "black boxes," i.e., the internal workings of a component are not a consideration in the construction of a Durra application.

A Durra programmer describes an application as a collection of *processes* (instances of Durra task descriptions) connected to each other in a graph structure via *links* (instances of channel descriptions). Lower level components are used as building blocks for higher level task descriptions. Application descriptions are simply compound task descriptions that describe a complete application.

A component's input/output interface is specified by the *ports* section (see Figure 2-1) of its description. Ports are named, unidirectional, locally-defined conduits through which processes may transmit/receive data. Ports have a Durra data type associated with them to allow semantic checking of intercomponent port connections.

Durra data type declarations define either a *size* type or a *union* type. A size type declaration associates an identifier with a data size (or a range of data sizes) expressed in bits. A union type declaration defines a new type as the union of one or more previously declared types. This type concept is analogous to our "black box" treatment of tasks--no semantic information other than the type name and the size of the data is derived from a type declaration. Here are some examples of Durra type declarations:

```
type byte is size 8;
type scalar is size 4*sizeof(byte);
type message is union (byte, scalar);
```

[2] Throughout this document, the term *task* refers to a generalized "thread of control" concept rather than to the analogous Ada construct, except where noted.

[3] The actual Ada code that implements a Durra task may, in fact, be a multitasking program. However, from the Durra perspective the program is a single thread of control.

```
task taskname (parameter-list)
        -- Values for the parameters are provided in task selections
        ports
           port-declarations
           -- A description of the input-output interface of the task
        behavior
           specification-list
           -- Labelled formal specifications of the behavior of the task
        attributes
           attribute-value-pairs
           -- A list of additional properties of the task
        components --(for compound tasks only)
           component-declarations
           -- A list of task and channel selections
        structures --(for compound tasks only)
           component-connection-structure
           -- A list of component connections
        reconfigurations --(for compound tasks only)
           condition-transition-pairs
           -- A list of conditional structure changes
        clusters --(for compound tasks only)
           cluster-component-associations
           -- A list of named physical groupings of components
end taskname;
```

**Figure 2-1: A Template for Task Descriptions**

The *behavior* section of the component description includes zero or more formal specifications of the behavior of the component's actual implementation. These specifications are not interpreted by the Durra compiler directly, but by associated tools. Although behavioral specifications are not part of the Durra language, a Durra component description provides a convenient placeholder for such specifications. Component descriptions containing behavioral specifications may then be used as components of an application description. A specification analysis tool is thus provided with a framework for reasoning about the composition of the specifications within an application architecture.

The *attributes* section defines additional properties of the component, such as version number or type of processor required. A primitive task description must be associated (via an attribute value) with a specific Ada procedure that is its implementation.

Figure 2-2 is an example of a primitive Durra task description. The task *producer* has one output port for data of type *message*. It is intended to run on a Sun4 processor, and its implementation is the Ada procedure “producer” in the library “/usr/durra/srclib”.

```
task producer
  ports
    output : out message;
  attributes
    processor = “sun4”;
    procedure_name = “producer”;
    library = “/usr/durra/srclib”;
end producer;
```

**Figure 2-2: A Primitive Durra Task Description**

A channel description is always primitive and is associated with a specific Ada package that implements it. Channels are intermediary processes which control the flow of data between user processes. Channel implementations for many frequently used communication disciplines are provided as part of the Durra support environment. These include FIFO and priority queue, broadcast, and merge, among others.

Both task and channel descriptions may be parameterized to allow for more flexible use of components. For example, one instance of a broadcast channel may be defined to have 3 output ports and another instance to have 10 output ports. Figure 2-3 contains descriptions of a generic channel (*fifo*) and a generic task (*consumer*). Each has a formal parameter that determines the data type of messages it can handle. The *buffer_size* parameter for the *fifo* channel specifies the number of messages that can be buffered by each input port of the channel. The *code* parameter for the *consumer* task specifies the Ada unit that implements the task for the given data type. Parameter values are supplied by *task/channel selections.* Selections are templates (identical to primitive description templates) that are used in compound task descriptions to select lower level components with the desired properties.

A compound task description must include additional information about its structure. Its component processes and links are defined in its *components* section (see Figure 2-1) and the manner in which they are logically connected (which may vary dynamically) is specified in its *structures* section. If the structure of the compound task is allowed to vary, then there must be a *reconfigurations* section that describes a set of structural changes and the conditions under which the changes will occur. The *clusters* section specifies the physical grouping of components into executable images, which may well be orthogonal to the logical connections described in the *structures* section.

Figure 2-4 is an example of a compound task description. The task name *consumer2* was chosen to avoid confusion in this presentation; however, we could have overloaded the name *consumer* from Figure 2-3 without conflict since the two tasks have different parameter and port profiles. *Consumer2* defines two internal processes, each of which is an instance of the previously defined *consumer*. The process declarations in the *components* section are examples

```
channel fifo(msg_type: identifier,buffer_size: integer)
  ports
    input: in msg_type;
    output: out msg_type;
  attributes
    processor = “sun4”;
    bound = buffer_size;
    package_name = “fifo_channel”;
    library = “/usr/durra/channels”;
end fifo;

task consumer (msg_type:identifier,code: string)
  ports
    input: in msg_type;
  attributes
    processor = “sun4”;
    procedure_name = code;
    library = “/usr/durra/srclib”;
end consumer;
```

**Figure 2-3: Generic Channel and Task Descriptions**

```
task consumer2
 ports
   in1: in byte;
   in2: in scalar;
 components
   c1: task consumer(byte, “byte_consumer”);
   c2: task consumer(scalar, “scalar_consumer”);
 structure
   L1: begin
         baseline c1, c2;
         bind  in1 = c1.input,
               in2 = c2.input;
       end L1;
end consumer2;
```

**Figure 2-4: A Compound Task Description**

of task selections that supply arguments to bind values to the formal parameters of the *consumer* task description. The *structure* section in Figure 2-4 is very simple. In Durra, the structure of an application is described as a collection of labelled *configuration levels*, which may be either nested or independent. There is only one configuration level (*L1*) in this application description. The *baseline* statement defines which processes and links are active at a given level. The *bind* statement defines a binding between the external ports presented by the interface of *consumer2* and the ports of the internal *consumer* processes. Since the internal processes do not communicate between themselves, no link declarations are required.

In Figure 2-5 we provide a Durra description of the classic producer-consumer problem as an example of a compound task description which also happens to be an application description. The building blocks for the task *producer_consumer* are the primitive components identified in Figure 2-2 and Figure 2-3. Since this is a top-level description, there are no external ports to

```
task producer_consumer
 components
  p: task producer;
  c: task consumer(message,  "message_consumer");
  buffer: channel fifo(message,10);
 structure
  L1:  begin
        baseline p, c, buffer;
          buffer: p.output >> c.input;
       end L1;
 clusters
  cl1 : p, buffer;
  cl2 : c;
end producer_consumer;
```

**Figure 2-5: A Producer-Consumer Application Description**

bind, but we must establish the connections between the processes that are defined internally. Connections are expressed in terms of the link implementing the connection. Thus, the link *buffer* connects the port *p.output* to the port *c.input*. The Durra compiler ensures that all ports are connected and that they are connected to ports of the proper data type and direction.

The Durra programmer specifies the distribution of application components by assigning them to virtual nodes called *clusters*. The *clusters* section of the description specifies that process *p* and link *buffer* will be physically grouped together at runtime, but process *c* will be linked into a separate executable program. This concept will be discussed in more detail in Section 2.2.2.

We require a more complex application description in order to demonstrate Durra's ability to express dynamic reconfiguration requirements.Figure 2-6 is an extension of the description in Figure 2-5. Two new components have been added: an instance of *consumer2* and an instance of a channel which implements a deal-by-type discipline. We omit the description of *deal_bt_channel* here to conserve space. This channel accepts input of a generic type and deals the input to a receiver requesting that type of data. The *structures* section in this example has been expanded to include a second configuration level, *L2*, which is nested within level *L1*. This level incorporates the two new components, *c2* and *dealer*, and excludes two of the components from *L1*, *c* and *buffer*. Since process *p* is not explicitly excluded from the nested configuration description, it survives into the new configuration. The port *p.output* is reconnected to link *dealer* and the two input ports of *c2* are associated with the two output ports of *dealer*. Note that this is not a data type conflict since the ports of *dealer* are defined to be of type *message*, which is the union type encompassing the types *byte* and *scalar*, the types of ports *c2.in1* and *c2.in2*, respectively.

```
task dynamic_producer_consumer
 components
  p: task producer;
  c: task consumer(message, “message_consumer”);
  c2: task consumer2;
  buffer: channel fifo(message, 10);
  dealer: channel deal_bt_channel(2,message);
 structure
  L1: begin
        baseline p, c, buffer;
        buffer: p.output >> c.input;
    L2: begin
          include dealer, c2;
          exclude c, buffer;
          dealer: p.output >> c2.in1, c2.in2;
        end L2;
      end L1;
 reconfigurations
  enter => L1;
  L1 => L2 when signal(c, 1);
 clusters
  cl1 : p, buffer, dealer;
  cl2 : c, c2;
end dynamic_producer_consumer;
```

**Figure 2-6: A Producer-Consumer Application Description Featuring Dynamic Reconfiguration**

The *reconfigurations* section of the *dynamic_producer_consumer* application description prescribes the conditions under which the configurations specified in the *structures* section shall be entered. Transition from one configuration to another is indicated by a configuration name pair on opposite sides of an arrow operator. When the application is in the configuration on the left-hand side of the arrow, the application is eligible to reconfigure to the configuration on the right-hand side of the arrow. A condition is usually associated with the transition, as in the transition from *L1* to *L2* in Figure 2-6. In this particular case, the transition will occur when the Durra runtime receives a *signal* (see Section 2.3.4) from process *c*. Durra assigns no semantic content to particular signal values; the interpretation of such signals is a function of the application description. The transition to configuration *L1* is a special case-- *L1* is to be entered unconditionally at application start-up.

## 2.2 The Durra Application Development Support Environment

### 2.2.1 Compilation and Libraries

Compiled Durra descriptions are maintained in libraries in an intermediate attributed syntax tree form. A Durra library may have multiple ancestor Durra libraries from which previously compiled descriptions are inherited. A library manager tool is used to create and manipulate these libraries.

Compilation of a primitive task/channel description results in a library entry which contains a reference to the Ada unit that is specified as the implementation of the description. For a compound Durra task description, the compiler uses the information provided in each task or channel selection to pick a component description that satisfies the selection requirements from a Durra library. If more than one component satisfies the requirements, the compiler picks the one most recently entered in the library (and warns the user about the ambiguity). The compound description is then entered in the library with pointers to the library entries for its component descriptions. Figure 2-7 shows graphically the relationships between the tools, libraries, and environment described here and in the following sections.

### 2.2.2 Code Generation

If the compound description is a complete application description, then the Durra compiler can generate an Ada package body for each cluster defined in the application. The package body imports the implementations of the components assigned to the cluster, creates Ada tasks to serve as threads for the implementations, generates code to evaluate reconfiguration conditions (if any are specified), and defines an Ada task that modifies the cluster structure as required. All these activities are specific to the cluster for which the package body is generated. The package body also defines a set of tables, common to all clusters in the application, that describe the complete application structure. A hardware configuration table, which defines the environment in which the distributed application will run, is optionally included. (If not included here, it must be specified at runtime.) As part of the support environment, we provide a set of runtime support packages that are used by all clusters. These support packages, the cluster-specific package body, and a driver subprogram are combined into a single Ada program that implements the cluster.

### 2.2.3 Support for Prototyping Applications

In general, the Ada implementation associated with a Durra task description will be hand-coded. However, for prototyping purposes, we provide two tools that interpret a particular type of behavioral specification called *timing expressions*. Timing expressions specify the duration of and intervals between port operations that the finished component is expected to demonstrate. One of the tools is an emulator that mimics any such specification; the other generates an Ada procedure that mimics a specific timing specification. The emulator or the generated procedure can be specified as the “implementation” of the component and imported by a cluster like any other user procedure.

We anticipate the development of additional tools to support emulation or code generation from other types of formal behavioral specification. We are currently working with Hughes Aircraft Company to develop a generator for a language based on restricted activity and data graphs [2]. These graphs may be used to define the flow of control and data within a Durra task as well as its logical, timing, and resource constraints. The Specification Methodology for

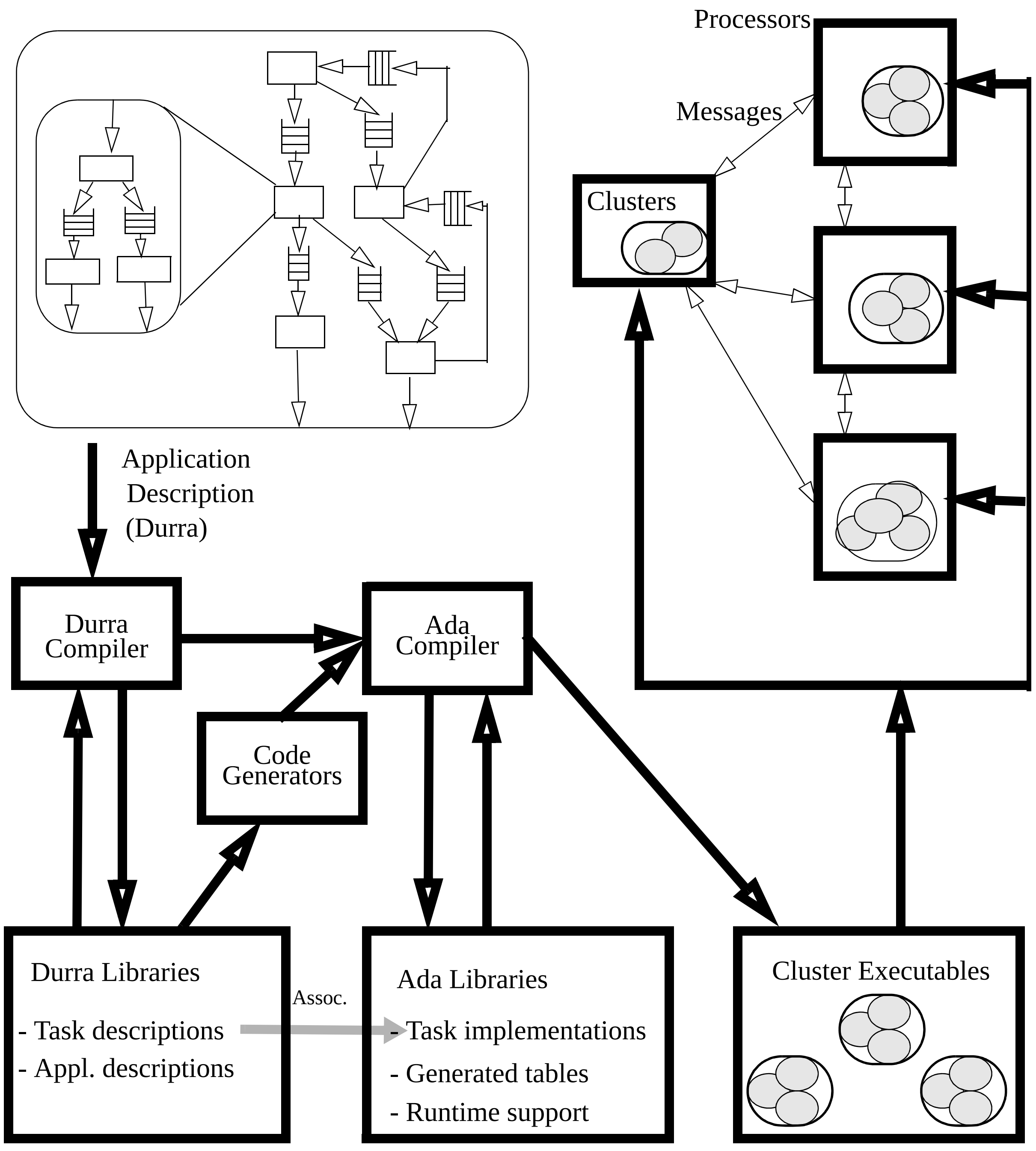

**Figure 2-7: Application Development Scenario**

Adaptive Real-Time Systems (SMARTS) describes how these graphs may be used in conjunction with the Durra language to specify the software architecture of highly adaptive real-time systems. In particular, this methodology defines a strategy for implementing dynamic task priorities using the facilities of the Durra language.

### 2.2.4 Version Control

We rely on vendor-supplied Ada library management tools to ensure version control for the Ada implementations associated with the Durra components. Since we are working in a Unix environment, we support version control of Durra application descriptions through automatic generation of a “make” file for each application. This is made possible by description dependency information maintained in the Durra libraries. The “make” file also coordinates the two control activities, ensuring that regeneration of the clusters due to changes in the Durra application description always uses the appropriate Ada units.

## 2.3 The Durra Runtime Environment

The Durra runtime environment:

- Establishes communications between clusters.
- Starts and terminates Durra processes and links.
- Transports data between Durra processes and links.
- Evaluates reconfiguration conditions.
- Performs reconfigurations.

The Durra runtime has no responsibility for scheduling Durra processes and links other than starting and terminating them. Since they are implemented by Ada tasks, they are scheduled by the Ada runtime and, where applicable, the host operating system scheduler. The priority of a Durra process can be passed to the Ada runtime via a “priority” attribute in the task description.

### 2.3.1 Model of Interprocess Communication

Durra implements a *buffered message passing* model of interprocess communication. Recall from Section 2.1 that Durra channels may have a “bound” attribute. The value of this attribute determines the size (in number of messages) of the buffer associated with each input port of the channel. When a producer process attempts to send data to a channel, the producer will block (i.e., be suspended) if the buffer associated with the port is full. Conversely, a consumer process attempting to read data from a channel will block if the buffer is empty.

Application designers thus have control over the flow of data between processes. Setting the buffer bound to zero forces synchronous communication, since either process will block until the other arrives. For practical purposes, one can achieve asynchronous communication by setting the bound to a very large number.

It should be noted that a Durra process may have additional input/output capabilities beyond its Durra ports. An example is reading data from a file; this can be modelled with a Durra channel, but it is not strictly necessary.

### 2.3.2 The Durra Runtime Interface

An Ada procedure that implements a Durra task accesses Durra runtime services via the *Durra_Interface* package. Figure 2-5 lists the services available to the Ada programmer. A pro-

```
Finish
 -- Inform the Durra runtime that the process is preparing to terminate.
Get_Application_Time
 -- Get elapsed time since this application entered its initial configuration.
Get_Attribute
 -- Get value of an attribute defined in the Durra description of this process.
Get_Port
 -- Get data from a Durra port.
Get_PortID
 -- Obtains a handle for a Durra port name.
Get_Process_Time
 -- Get elapsed time since this process started.
Get_TypeID
 -- Obtains a handle for a Durra type name.
Initialize
 -- Obtains a Process_ID, a handle for further runtime service requests.
Raise_Signal
 -- Send a signal to the runtime.
Safe
 -- Indicate that it is safe to perform a reconfiguration involving
  -- this process.
Send_Port
 -- Send data to a Durra port.
Test_Input_Port
 -- Returns the number of messages now available on this port, as well as
 -- the type and size of the next message that will be delivered.
Test_Output_Port
 -- Returns the number of messages that the process is guaranteed to be
 -- able to send without blocking.
```

**Figure 2-8: Durra Runtime Services**

cedure will typically begin by calling *Initialize* and then make one or more calls to *Get_PortID* and *Get_TypeID*. This establishes the presence of the process with the runtime and provides the process with the Durra object handles necessary for interprocess communication. It can then use the *Send_Port* and *Get_Port* calls to transmit and receive data. The *Test_Input_Port* and *Test_Output_Port* calls may be used to test the status of the port in order to avoid blocking when sending and/or receiving. When the procedure has completed its work it must call *Finish*; failure to do so causes unpredictable application behavior.

The implementation of the port-related calls varies depending on the distribution of the processes and the link involved in the transaction. In Figure 2-9 we see an example of two different kinds of interprocess communication. The gray boxes in the figure represent Ada tasks. In Cluster 1, user processes P and Q are communicating through link A. Process P is also communicating with remote user process R through remote link B. Links are passive tasks; they

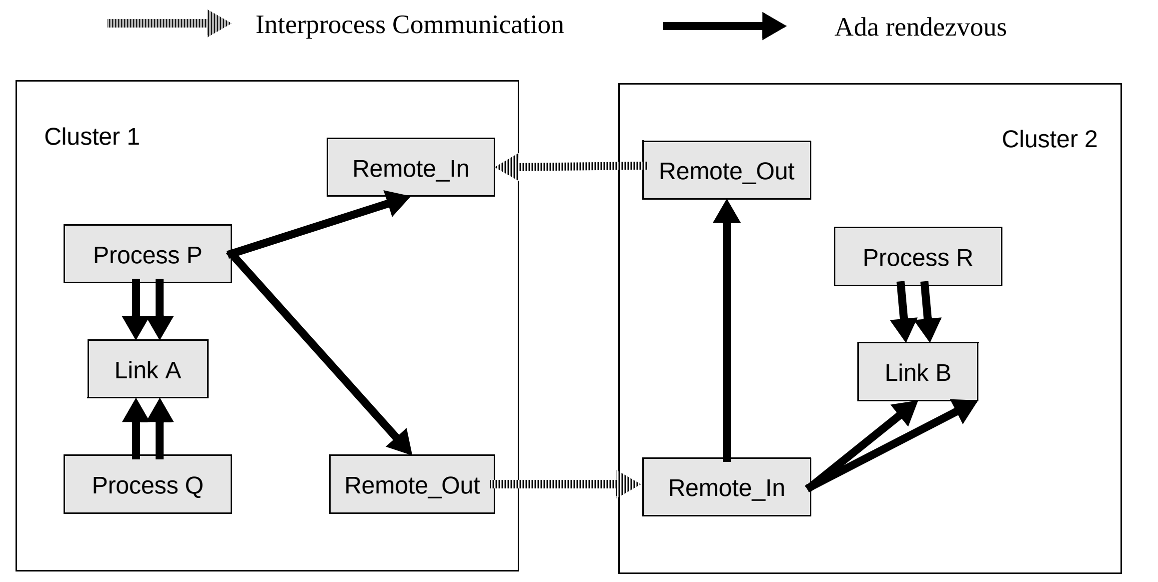

**Figure 2-9: Interprocess Communication in the Durra Runtime**

are servers for requests from processes. So in order for process P to send data to process Q, both must rendezvous with link A. If P is blocked while attempting to send data, then its rendezvous must be ended to allow link A to process other requests. Process P must then issue a second entry call to wait for notification that the buffer is no longer full. Similarly, process Q will block if the buffer is empty. The rendezvous must be released and process Q must issue a second entry call and wait for data to arrive. So if all three processes are local, a minimum of two and a maximum of three rendezvous are required to transfer the data. For non-local communications, each cluster uses a pair of Ada tasks, *Remote_Out* and *Remote_In*. In order for process P to send data to process R through link B, a total of six or seven rendezvous are required, as well as two network messages. Although the work load is distributed, this is obviously a high overhead for message delivery and we are currently examining ways to reduce it.

### 2.3.3 Clusters

The result of the compilation process described earlier is an executable program for each cluster named in the application description. A wide range of distribution choices is available to the programmer, from assignment of all components to a single cluster (the degenerate case, in which the application is not actually distributed), to assignment of a single component per cluster. The clusters themselves may then be assigned to processors in various combinations ranging from all on one processor (in multiprocessing environments) to one cluster per processor.

To avoid starting all the clusters of a Durra application independently, a tool external to the Durra runtime may be required. In the UNIX environment, we use a program called the *Durra_Launcher* to start all clusters except the first, which is started by hand. This cluster is considered the *master*, a first among equals. The master cluster is then responsible for instructing the *Durra_Launcher* to start all the other clusters. Once started, a cluster attempts to establish communications with the other clusters in the configuration by means of an established protocol.

Mastership is a dynamic status that may be passed among clusters as the structure of an application evolves. The master differs from the other clusters primarily in its responsibility for control of the reconfiguration process. There are two reasons for the designation of a master. The first is that a reconfiguration condition expression may be composed of subconditions which are detected in separate clusters. A logical conjunction of these subconditions must be evaluated in either a single cluster or in all clusters. We have chosen a single master cluster since this requires less intercluster communication. For example, assuming the structure shown in Figure 2-9, the subexpressions in the condition “signal(P,1) and signal(R,2)” would be detected in Cluster 1 and Cluster 2 independently. The evaluation of the complete expression would be performed in the cluster designated as master. The master should therefore be designated as the cluster where reconfiguration condition evaluation can be done most efficiently (locally). This may vary at different configuration levels, so the master designation is assignable. Once an entire reconfiguration condition evaluates to true, the reconfiguration must be initiated. That is the second reason for the single master approach. One cluster must control the initiation of reconfigurations in order to avoid concurrent incompatible reconfigurations. The master cluster notifies the other clusters when a reconfiguration is beginning. Then the reconfiguration is carried out in parallel by the individual clusters, with each cluster responsible for the changes that affect its local component set. In case of failure of the master cluster, the surviving clusters must agree on a new master. User components in the failed cluster can be restarted on a new cluster, but their computation states will be lost unless the application has provided some means of recovering them.

As mentioned earlier, the hardware configuration on which the distributed application will run must be specified either at compile time or at runtime. The hardware configuration information is contained in a file. If this file is supplied to the Durra compiler when Ada code is generated, then the information can be included in the cluster-specific tables being generated. If supplied at runtime, the file must be read and the tables modified. In either case, the specification of hardware resources must be complete; Durra does not support dynamic reconfiguration of the application platform. Hardware specification at compile time decreases cluster initialization overhead at the cost of experimental flexibility. The reverse is true of runtime specification. Since the runtime specification overhead is small and the cost of code regeneration and recompilation can be high, the likely best choice is to use runtime specification during development or experimentation and compile time specification for a production system.

### 2.3.4 Dynamic Reconfiguration

Dynamic reconfiguration of an application is the modification of its structure while the application is running. This may involve addition or subtraction of processes, or simply redistribution of the existing processes. Reconfiguration in the Durra world does not involve process migration. A task that is expected to run in two different clusters during the lifetime of an application must be declared as two separate process components.

A Durra reconfiguration condition is a Boolean expression involving information available to the clusters at runtime as well as signals from application processes. If reconfiguration conditions are present in the application description, then part of the code generated for each cluster is a case statement that allows evaluation of the conditions enabled at each configuration level. Conditions based on elapsed time are implemented as delay statements; when the delay expires, the condition is true. When an expression evaluates to true, the case statement prescribes an entry call to the *State_Changer* task, which is also generated by the Durra compiler. The master cluster passes the information to the other clusters. In each cluster, *State_Changer* carries out the reconfiguration, starting new tasks and terminating old ones as required to transition to the new configuration.

## 2.4 Application Issues

### 2.4.1 Programming Language Restrictions

Most Durra tasks will be implemented in the Ada programming language. In some Ada environments, one may be able to incorporate procedures written in other languages as Durra tasks. To do this, one has to have an environment that allows for Ada to interface to the other language and that also provides a mechanism to call Ada subprograms from the other language so that the foreign procedure can call Durra runtime services.

In our original implementation of Durra, each user component was mapped onto an operating system process. The Durra runtime support for each processor was also in a separate process. An advantage of this approach is the ability to support mixed-language applications easily; one only needs an implementation of the Durra runtime interface for each language to be used. There are several disadvantages associated with this approach, though. One is the cost of interprocess communication. In applications where multiple Durra processes are assigned to a single processor, it is more efficient to model the concurrency using Ada tasking within a single operating system process. The original Durra runtime was also less portable; since it assumed a multiprocessing operating system environment, it could not be ported to bare machine targets. Because we felt these disadvantages outweighed the value of language independence, we made a conscious decision to provide only minimal support for mixed language Durra applications.

### 2.4.2 Support for Heterogenous Machine Architectures

Durra allows distribution of a single application across some number of physical processors. There is no requirement that these processors be homogeneous. We have run small experimental applications that were distributed across Sun4, Sun3, VAX/ULTRIX, and VAX/VMS hosts in various combinations. Of course, each processor in any potential Durra platform must have a validated Ada compiler available. Since the components of a Durra application are standard Ada programs, there are no special requirements on the Ada compiler or runtime. The Durra runtime library must be ported to each target architecture in the platform and all the processors must support a common communication protocol. We have used the TCP protocol in our experiments to date.

Data representation issues are not addressed by the Durra runtime. This is a natural result of Durra's treatment of component tasks and types as "black boxes." Durra tasks do not know what happens to data once it has been sent to a port. Therefore, it is up to application designers, who do know about the target architectures and the distribution of processes, to account for required data representation changes in their designs. This can be done by either inserting additional processes or special purpose links that transform the data as they transfer it.

### 2.4.3 Application Domain Independence

The Durra language and runtime environment are domain-independent. They support the development of applications consisting of distributed, message-passing components. The nature of the components and messages is a domain-specific concern, above Durra and its implementation. For example, one could implement a distributed programming environment in which various tools (compiler phases, library managers, etc.) execute as cooperating Durra tasks, sending various kinds of data structures (annotated syntax trees, module dependency lists, etc.) through channels which provide the appropriate data transformation and filtering operations. Both tasks and channels could be user written or automatically generated by compiler-compilers or similar tools using formal language specifications and interface description languages such as IDL[3]. These domain-specific tools are outside the scope of our project.

# 3 Related Work

There are two distinct areas of ongoing research to which Durra is related. One is support for distribution of Ada programs in particular. The other is programming languages for the specification and prototyping of large-grained parallel applications in general.

Numerous projects are underway to develop tools and methodologies for the support of distributed Ada software. These attempts fall into two broad classes: either the application is written as a single Ada program and then partitioned for distribution, or the application is written as multiple programs which communicate through some standardized interface. Examples of the former include APPL [4], Aspect [5], and DARTS[6]. Examples of the latter include DIADEM [7] and DARK [8]. Single program approaches have many advantages, including communication via standard Ada facilities and type and consistency checking enforced by the language. However, APPL requires extensive compiler support which makes distribution to heterogenous processor environments problematic. In DARK and DIADEM, separate program components communicate via a remote procedure call and a remote rendezvous mechanism, respectively. DARK does not provide language support for distribution specification, so application structure is not separated from function. The virtual node approach taken in DIADEM is similar to ours, but DIADEM allows for compile-time checking of interfaces like single program approaches. Durra's advantages over all these approaches are its language support for reconfiguration and component reuse in multiple application environments, and its provision of significant flexibility (via user-defined channels) in the forms interthread communication may take (asynchronous versus synchronous, FIFO versus priority message arrival, etc.).

The three systems most like Durra are NAS[9], CSL/Model [10], and Conic [11]. NAS, which is being used in at least one fielded software product, is probably the most mature technology in this arena. Its distributed application structure model is very similar to Durra's. NAS provides operator interfaces in support of performance/error monitoring and operator-controlled dynamic reconfiguration. However, Durra's method of software architecture specification and its application program runtime interface is simpler than that of NAS. CSL is used to specify application interconnection and distribution in the same manner as Durra. A CSL application can incorporate implementations generated by the Model compiler from behavioral specifications. CSL's support for reconfiguration and communication flexibility is limited, though. Conic's constructive approach is similar to Durra's but since Conic applications must be written in Pascal, their virtual nodes cannot be multithreaded. Unlike Durra, both CSL and Conic offer graphical front ends to their respective specification languages.

Polylith [12] and Reality [13] are highly flexible distributed application description languages focused on support for prototyping. Polylith provides a more flexible approach to dynamic reconfiguration than Durra, but it requires user intervention. Polylith modules are connected via a software bus, to which a user can attach new modules at arbitrary times. A framework for application-controlled reconfiguration is under development[14]. Reality is a more ambitious project; its long-term goals include facilitating the evolution of prototypes to production quality software/hardware systems.

# References

[1] Barbacci, M.R., D.L. Doubleday, C.B. Weinstock, M.J. Gardner, J.M. Wing. *Durra: A Task-Level Description Language Reference Manual (Version 3)*, SEI Technical Report CMU/SEI-91-TR-18, December, 1991, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, Pa.

[2] Muntz, A.H., and R.W. Lichota. "A Requirements Specification Method for Adaptive Real-Time Systems," to appear in *Proceedings of the IEEE Real-Time Systems Symposium,* December, 1991.

[3] Nestor, J. R., J. M. Newcomer, P. Giannini, and D. L. Stone. *IDL: The Language and Its Implementation,* Prentice Hall 1990. ISBN 0-13-450214-0.

[4] Jha, R., and G. Eisenhauer. "Distributed Ada - Approach and Implementation," *Proceedings of TRI-Ada '89*, Pittsburgh, Pa., October 23-26, 1989, pp. 439-449.

[5] Hutcheon, A.D., and A.J. Wellings. "Supporting Ada in a Distributed Environment," *Proceedings of the Second International Workshop on Real-Time Ada Issues, Ada Letters,* Vol. VIII, No. 7, Fall 1988, pp. 113-117.

[6] Wengelin, D., and L. Asplund. "Application of Ada on a Distributed Missile Control System", *Proceedings of TRI-Ada '90,* Baltimore, Md., December 3-7, 1990, pp. 300-305.

[7] Atkinson, C., and S. J. Goldsack. "Communication between Ada Programs in DIADEM," *Proceedings of the Second International Workshop on Real-Time Ada Issues, Ada Letters,* Vol. VIII, No. 7, Fall 1988, pp. 86-96.

[8] Bamberger, J., C. Colket, R. Firth, D. Klein, R. Van Scoy. *Kernel Facilities Definition, Distributed Ada Real-Time Kernel Project.* Technical Report CMU/SEI-88-TR-16, DTIC: ADA198933, July 1988, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, Pa.

[9] Royce, Walker. "Reliable, Reusable Ada Components for Constructing Large, Distributed Multi-Task Networks: Network Architecture Services (NAS)," *Proceedings of TRI-Ada '89*, Pittsburgh, Pa., October 23-26, 1989, pp. 500-516.

[10] Shi, Y., and N. Prywes. "Generating Multitasking Ada Programs from High-Level Specifications", *Proceedings of the Third International Conference on Ada Applications and Environments*, Manchester, N.H., May 23-25, 1988, pp. 137-149.

[11] Kramer, J., and J. Magee. "A Constructive Approach to the Design of Distributed Systems", *Proceedings of the 10th International Conference on Distributed Computing Systems*, Paris, France, May 28-June 1, 1990, pp. 580-587.

[12] Purtilo, J.M., and P. Jalote. "An Environment for Prototyping Distributed Applications", *Proceedings of the Ninth International Conference on Distributed Computing Systems*, Newport Beach, Calif., June 5-9, 1989, pp. 588-594.

[13] Belz, F.C., and D.C. Luckham. “A New Approach to Prototyping Ada-based Hardware/Software Systems,” *Proceedings of TRI-Ada ‘90*, Baltimore, Md., December 3-7, 1990, pp. 141-155.

[14] Purtilo, J.M., and C.R. Hofmeister. “Dynamic Reconfiguration of Distributed Programs,” *Proceedings of the 11th International Conference on Distributed Computing Systems*, Arlington, Texas, May 20-24, 1991, pp. 560-571.