Supporting distribution and dynamic reconfiguration in AdaPT

To cite this article: A B Gargaro et al 1994 Distrib. Syst. Engng. 1 145

View the article online for updates and enhancements.

Related content
- Distributed and parallel Ada and the Ada 9X recommendations
  R A Volz, R Theriault, R Waldrop et al.
- KTK: kernel support for configurable objects and invocations
  B Mukherjee, D Silva, K Schwan et al.
- Towards an integrated approach to fault tolerance in Delta-4
  P A Barrett and N A Speirs

Recent citations
- Distributed and parallel Ada and the Ada 9X recommendations
  R A Volz et al.
Supporting distribution and dynamic reconfiguration in AdaPT

A B Gargarot, S J Goldsack, A A Holzbacher-Valero, R Volz, R Waldrop and A J Wellings
(Computer Sciences Corporation, Moorestown, NJ, USA)
(Department of Computing, Imperial College, London, UK)
(LIX, Ecole Polytechnique, Paris, France)
(Department of Computer Science, Texas A&M University, College Station, TX, USA)
(Department of Computer Science, University of York, UK)

Received 5 September 1993

Abstract. It is widely accepted that Ada83 provides inadequate support for the programming of distributed systems. Ada9X has introduced a unit of distribution called a partition. Partitions comprise aggregations of library units that collectively may execute in a distributed target execution environment. Each partition corresponds to a single execution site where all its library units occupy the same logical address space. The principal interface between partitions is one or more package specifications.

Although Ada9X provides basic support for partitioning applications, it falls short of providing the full expressive power that would be expected of a language specially designed to support distributed systems. In particular, the unit of distribution is not a first class language object and methods for dynamic configuration are primarily left to the implementor.

In this paper we briefly review the main requirements for programming distributed systems and illustrate where the current Ada9X proposals lack expressive power. We then introduce a new variant of Ada called AdaPT which has been designed to address explicitly partitioning and dynamic reconfiguration of distributed Ada programs. We illustrate how AdaPT programs can be written and detail their translation into Ada83 and Ada9X.

1. Introduction

The use of computers that are embedded in some wider engineering application is increasing rapidly. Such systems typically have a number of common characteristics: they must respond to externally generated input stimuli within a finite and specified period; they must be extremely reliable and/or safe; they are often geographically distributed over both a local and a wide area; they may contain a very large and complex software component; and, they may contain processing elements which are subject to cost, size or weight constraints.

The programming abstractions necessary to express these characteristics for embedded computer applications are unavailable in most of today's programming languages. The Ada programming language was designed to fill this need and provide support for embedded applications within a unified language framework. However, it is now commonly accepted that the language has not totally succeeded in achieving all its stated design goals. While Ada (as defined by ISO 8652:1987), with an appropriate project support environment, has successfully addressed many of the software engineering issues associated with the production of large real-time software, it has failed to satisfy applications requiring the use of multiple computers or parallel-intensive computation. A particular concern is the language’s lack of support for the construction of fault tolerant distributed systems—although there is much controversy within the Ada community as to how this support should be provided [18, 39, 40].

Despite references in the Ada Language Reference Manual to the possible parallel execution of Ada applications, attempts to develop techniques for fault tolerant distributed and parallel execution have met with only moderate success. Most have revolved around the concept of the Virtual Node [20], which is a software component intended to serve as a unit of allocation in a network. Virtual nodes provide strongly cohesive components, each guaranteed to run in a single physical node (where a physical node is one or more processing units having access to common memory). They interact with other virtual nodes forming the complete system only by message passing. While virtual nodes have been successfully used to construct Ada distributed systems, the solutions offered by the various research projects all suffer from limitations arising from the nature of the language.
particular, the structures used are awkward and unable to provide the capability of dynamic reconfiguration following failure which is one of the main purposes of distributing software in many embedded, safety critical applications. Reviews of these attempts may be found in [3, 8, 20].

To deal with these, and other limitations of Ada, a language revision process, commonly called Ada9X, is in progress. This activity is now reaching its conclusion and it is possible to see the scope of the proposed changes. Overall the number of changes is large [35] (including the introduction of object-oriented programming facilities, better support for real-time systems, and more efficient programming in the large abstractions); however, the support for distribution has been conservative.

In this paper we briefly review the main requirements for programming distributed systems. We then summarize the Ada9X model and illustrate why we believe that it has failed to meet some of these requirements. We then propose a new model called AdaPT [19], Ada with Partitions, which was developed in parallel with Ada9X [15, 16] and which overcomes some of the limitations of the Ada9X model. We present an example application, and then discuss how AdaPT can be translated in Ada83 and Ada9X.

The goal of this paper is not to undermine the current Ada9X process which we believe has produced an elegant solution within the allowed scope for changes. Rather our aim is to explore its limitations, illustrate how they can be overcome with more ambitious language changes, and show how these might be implemented in Ada83 or Ada9X.

2. Distribution issues

Most distributed applications in the real-time domain have both functional (application-level) and non-functional requirements (sometimes called system-level requirements) imposed upon them. Functional requirements describe the processing that must be performed by the application: for example reading sensors and outputting values to actuators. Non-functional requirements, on the other hand, are concerned with the dependability [30] of the application (reliability, safety and security), the timeliness of the results it produces (sometimes expressed in terms of performance), configuration of the system, and the ability to upgrade the software dynamically in response to changing requirements [10]. Clearly the non-functional requirements can have a major impact on the way in which the functional requirements are satisfied.

Key issues in the design and implementation of real-time applications are the extent to which the implementation language supports both the functional and non-functional components and the extent to which these functions are separated [26]. Unfortunately, it is not always possible to separate completely functional and non-functional components. For example, an application may want to provide a degraded service following a reduction in the available processing power, or during a transient overload. Consequently, although separating concerns is generally desired it must still be possible for the functional and non-functional components to interact with each other.

There are a number of existing languages designed for the development of distributed systems. In general these languages provide a compromise between completely separating the functional and the distribution management components of the application and allowing the functional components to dictate distribution. Typically, they include syntactic structures which define a unit of distribution (the granularity of potential distribution) having properties similar to those of virtual nodes (see section 1). While units of distribution cannot themselves be distributed and must reside on a single physical node, usually, no mention is made of the actual mapping of virtual nodes to physical nodes in the distributed architecture—they reduce the freedom of the non-functional component of the application. Typically, a second 'tier' of programming, using a 'configuration language', that defines the populations and communication patterns of these 'virtual node' units in the network, is used [28]. An early example of a language with these properties is CONIC [29]. CONIC programs are composed of task modules, which correspond to virtual nodes, and group modules which cluster one or more task modules into larger structures intended to execute on the nodes in a network. Other such languages include Argus [31], SR [2, 9] and Durra [5] (which allows the functional components to be programmed in sequential Ada83, whilst providing its own mechanisms for communication, synchronization and reconfiguration). Darwin [32] attempts to go one stage further and provide a configuration language which is largely independent of the languages used to program the functional components.

Fault tolerance of hardware failure is becoming an increasingly important requirement for distributed systems. In general there are two ways in which a language or operating system can support fault-tolerant applications. The first is via the transparent replication of application-level processes on different physical nodes in the distributed systems (the nodes often have a particular failure property such as fail silent [34]). A range of techniques are available such as active replication [33], passive replication [6] and leader follower replication [7, 37]. Supporting such systems often requires complex atomic broadcast protocols [13, 27] which may not be easily analysable for their worst case performance. The second method of achieving fault tolerance is to provide a mechanism for the dynamic reconfiguration of a distributed program and to allow the programmer/designer to develop their own solutions. These mechanisms will often allow dynamic changes to programs thereby allowing bug fixes and upgrades due to changing requirements. It is this latter mechanism which is of interest to us in this paper.

3. The Ada9X model of distribution

3.1. Ada83 background

The Ada83 language alludes to supporting program distribution, however, although it does have enough
expressive power to program the functional components of an application it does not enable a clear separation between functional and non-functional concerns. This is because program units are not first class objects and configuration information is scattered (in the form of context clauses) throughout the program. Furthermore, there is no clear unit of distribution.

The Ada notion of a main program also causes problems for distribution management. It has been observed by Atkinson and Di Maio [4] that the Ada main program serves three simultaneous functions.

- It conceptually envelops the complete executing system.
- It is the unit formed by the compilation/linking system into the unique executable binary, representing the complete program as a single executable entity.
- It is the entry point at which the process of elaboration starts, following the invocation of the program by the operating system.

These features provide an appropriate model for constructing a single program for execution in a single physical node. For programs intended to execute in a network, they are not suitable. The first function is unnecessary, while the others need to be handled independently. For execution in a network, a collection of executable binary units must be developed, while the elaboration of these separate binaries must occur in a way that is controlled through the application, and may require concurrent elaboration of the separate network units.

3.2. Ada9X

The Ada9X model for programming distributed systems [1] specifies a partition as the unit of distribution. The model borrows from the virtual node concept and earlier work that developed the concept into a partition construct [17]. Partitions are not a first class language entities (in the sense that they cannot be declared as types and instances created). Instead they comprise aggregations of library units that collectively may execute in a distributed target execution environment. Each partition corresponds to a single execution site where all its library units occupy the same logical address space. The principal interface between partitions is one or more package specifications.

The partition model specifies a simple, consistent and systematic approach towards composing distributed systems. Partitions are specified subsequent to the compilation of their constituent library units. Programming cooperation among partitions is achieved through library level packages that allow access to data and subprograms in different partitions to which these packages are explicitly assigned. These library packages are identified at compile-time by categorization pragmas. In this way, strong typing and unit consistency is maintained across a distributed system, while avoiding the complexity of a distributed runtime system.

Partitions may be either active or passive. The library units comprising an active partition reside and execute upon the same processing node. In contrast, library units comprising a passive partition reside at a storage node that is directly accessible to the nodes of different active partitions that reference them.

A passive partition is assigned one or more shared passive packages. A shared passive package is a library package compiled with the pragma Shared_Passive and may depend only on other shared passive packages or declared-pure packages. Pure packages are restricted, e.g., they may not declare library level variables, so that they may be freely replicated in different active or passive partitions. There are restrictions on shared passive packages, e.g., they may not declare tasks. The restrictions ensure that the packages are pre-elaborated and are compatible with the properties of common address modules. The state of a shared passive package may persist between different executions of a partitioned program.

An active partition may name shared passive packages from passive partitions in its closure, thus referencing packages that are common to other partitions in the distributed system. Different active partitions (executing in separate nodes) may share protected data or call subprograms from such packages.

Active partitions may call subprograms in other active partitions. Calls to subprograms in a different active partition are allowed only if the called subprogram is referenced through a package that is compiled with the categorization pragma Remote_Call_Interface. The specification of remote call interface packages is restricted to preclude implicit remote references and may depend only on pure packages or other packages compiled with a categorization pragma. Each active partition calling the subprogram must name the corresponding remote call interface package in its closure. When an active partition calls such a subprogram, the call is termed a remote subprogram call. In addition, an asynchronous procedure call capability is provided to allow the caller and the called remote subprogram to execute independently once the call has been sent to the remote partition.

The predefined exception named Communication_Error is raised at the calling location whenever the called partition is (or becomes) inaccessible. An exception raised by executing the subprogram is propagated back to the calling partition. If the exception is not visible at the calling site, it may be handled by an others choices or be propagated to and handled by another partition. For asynchronous procedure calls, such exceptions are not propagated to the calling partition.

Ada9X supports dynamically bound subprogram calls which can be used to program more fault-tolerant paradigms. For example, through dynamically bound calls, a distributed program may reference subprograms in replicated partitions to safeguard against partition failures. In the event of a failure in a called partition, the calling partition redirects the call to a subprogram in a backup partition.

An advantage of dynamic binding is the relaxation of the requirement (of statically bound calls) for the calling partition to name in its closure the packages that explicitly declare the remote subprograms. Partitions need only name the remote call interface package that includes the
declaration of an appropriate general access (pointer) type. The utility of the dynamic binding capabilities are enhanced further when combined with a name server partition. Typically, the name server partition manages a repository of remote subprogram access value for the entire distributed system.

Finally, to achieve remote subprogram call communication using different style protocols, the notion of a Partition Communication Subsystem (PCS) is defined. The PCS is a layer of software specifying a common package interface (RPC) to which remote subprogram call implementations must comply. This allows a PCS to be implemented independently of a specific Ada9X compiler. The PCS may then connect to the appropriate communication protocol stack software.

Figure 1 diagrammatically illustrates how library units comprise an Ada9X partition.

3.2.1. Observations

- The Ada9X partition model provides support for distributed execution without introducing additional linguistic constructs. The motivation is to minimize the differences in programming distributed and non-distributed systems without bias towards specific distributed target architectures.
- Partitions elaborate and execute independently. The model does not provide for the Ada notions of time nor priority to span partitions. Normal packages elaborate in the partition in which they are referenced; if such packages have state then it is independent of the state of the same package elaborated in a different partition. For example, package Calendar when elaborated in different partitions is not required to maintain a synchronized system-wide clock.
- There is no requirement for a distributed Ada run-time system. Communication among active partitions is through passive partitions or through remote subprogram calls. The latter provides compatibility with in-progress standardization for communication among open systems applications [24] while allowing dynamically-bound calls that increase the range of distributed programming paradigms.
- A canonical model for implementing remote subprogram calls is specified through the requirement for all implementations to conform with the PCS interface package RPC_Support. This model supports both statically and dynamically bound remote subprogram calls in both synchronous and asynchronous forms.
- Implementations must define the required post-compilation support to construct partitions and to configure the partitions onto a distributed target system. The construction of partitions allows the explicit assignment of remote call interface packages and shared passive packages to partitions. The assignments must be consistent with the accessibility requirements of each partition's dependency graph. Consequently, the dynamic creation and replication of partitions cannot be expressed in writing portable programs.

3.2.2. Limitations of the Ada9X partition model. The most severe limitation of the Ada9X partition model is the absence of the ability to create and replicate partitions dynamically; such support may be provided by an implementation but reduces the portability of fault-tolerant distributed programs. In Ada9X, partition replication must be achieved statically by instantiating generic remote call interface or shared passive packages; each instantiation is assigned to a separate partition. The partition becomes the closure of the library units mentioned in the ‘with’ clauses of each instantiation. Unless remote access types are used, client partitions are required to reference the name of the instantiated package explicitly in their closure.

Various paradigms are available to ameliorate this limitation; for example, the notion of distributed objects has been proposed using the remote ‘access-to-class-wide type’ [17]. In this paradigm, the operations associated with dynamically created objects may be called remotely and are synchronized in the remote program with other calls to the same operation. Using distributed objects a degree of fault-tolerant programming can be achieved in the context of a simple client/server paradigm. A client partition may call synchronized operations in different server partitions through the access value of a dynamically created object that may be passed to one or more client partitions.

While this notion provides a more object-oriented approach to programming distributed paradigms, it fails to provide the flexibility and fault-tolerance provided by a more intuitive approach that supports the construction of partitions. The distributed objects must be created in the context of an existing partition; thus, the fundamental problem of statically composing the distributed program remains.
4. AdaPT—an alternative distribution model

One of the main constraints placed on the Ada9X language was that there should be no new language syntax to support abstractions for distributed programming (e.g., no new reserved words). This, in our view, severely limits the expressive power and ease of use of the language for supporting fault-tolerant distributed systems. In this section we describe an alternative model which overcomes these limitations.

The examination of all the above considerations in sections 2 and 3 has led to the development of several fundamental language constructs which we believe enable a clear separation between functional and non-functional (in our case distribution and fault tolerance) components and enable interaction between the two. The first, called a partition, is an abstraction for logically aggregating the functional program units for distributed execution. The second, a construct called a node, handles the configuring aspects of dynamically constructing a distributed system from partitions. The node is the program unit which is developed by the compilation and linking system into an executable binary unit for residency on an actual node of the network. It may encapsulate one or more partitions, as dictated by the system design. Partitions and nodes are supported by a new type of package, called a public package, which enables program types to be shared between partitions. Sections 4.1 and 4.2 discuss the details of partitions, nodes and public packages.

It is important to stress that the notion of an Ada program should be divorced from the notion of an executable binary unit. A distributed program will necessarily consist of many executable binary units. In our proposal we will introduce the notion of a distinguished node. The execution of an Ada program will consist of the creation of the distinguished node followed by its elaboration (see 4.2.2). The distinguished node will, possibly recursively, create and elaborate the other nodes in the program. Currently, we only allow a single distinguished node, although it might be possible to envisage more than one.

4.1. Participating a program

It is assumed that programs are decomposed by some design methodology (the specific methodology is not relevant here) into units that are to execute each on a single processor. Such units have been variously known as: guardians [31], resources [2], or, in the Ada community, virtual nodes [3, 14, 20, 36, 38]. In this paper they will be called partitions. Although composing programs from partitions has the disadvantage that it reduces the potential for distribution, it enables more efficient communication paradigms to be implemented between tasks which are guaranteed to share a physical address space [23].

To meet the distribution requirements, we give partitions the following characteristics, following the lead of early efforts toward distributing Ada programs [3, 11, 22].

- They will be units of modularity in a distributed system.
- They will be also the units of reuse—wherever possible programs should be composed from off-the-shelf partitions.
- They will provide well defined interfaces to other partitions in the system.
- They will encapsulate local resources. All access to these resources from remote partitions is via the partition interface.
- They will consist of zero or more tasks. As in any Ada program unit, the tasks within a partition may communicate with each other using shared memory. They will also be able to communicate with tasks in other partitions via the interfaces provided. This communication will be via procedure call or remote rendezvous.
- More than one partition will be configurable onto a single physical node. However, it is worth emphasizing that a partition will not be distributable among processors in the distributed system. Decomposing programs into partitions will therefore define the granularity of potential distribution of the application.

We associate a new library unit with a partition, which for the moment, we call the partition library unit. Strictly speaking the partition library unit is the external interface to the partition: the whole partition is represented by all the library units that are directly or indirectly 'wired' by the partition library (except for any other partition library units, which interface to other partitions). Figure 2 shows diagrammatically the structure of a partition. As partitions can only be referenced via the partition library unit, the

Figure 2. The structure of a partition.
A B Gargaro et al

Figure 3. An AdaPT partition.

Figure 4. One partition referencing another partition.

Figure 5. Sharing public library units between partitions.

term partition is used without ambiguity to indicate as required the partition library unit or the whole partition. Figure 3 shows a partition consisting of the partition library unit and three other library units.

Clearly in following the transitive closure of partition X another partition Y may appear in a context clause. In this case that branch of the closure on partition X is terminated, as Y represents the interface of another partition which is invoked by X. This is illustrated in figure 4.

If a (non-partition) library unit appears in two or more partitions then each partition has a separate instance of the library unit. For example if a library package, \( L \), appears in the context clause of two partitions then two distinct instances of the package \( L \) are created. Any types declared in package \( L \) are also distinct in the two instances. This is a departure from the current Ada language standard where it is only possible to have one instance of a library package (this departure has been adopted by Ada9X). The approach is also required to allow instances of the same partition to be created by the configuration tier of the program (i.e., partition types).

Communication between partitions occurs by one partition calling the interface of another partition (which is declared by the partition library unit). In order for two or more partitions to communicate effectively they must be able to share common type declarations. These type declarations are declared in another new type of library unit called a public unit. If two or more partitions name the same public library unit in their context clauses then that library unit is shared among the partitions. This is illustrated in figure 5.

4.1.1. Public library units. Public library units are equivalent to invariant-state packages. A public library unit is identified by the new keyword public and has a specification and may have a body. The interface to a public unit can contain items such as: types (but not access types), task types, task access types, subprograms and exceptions.

The body to a public unit must have a well defined constant state. The use of variables is precluded because the public unit must be capable of existing at all sites in the distributed system. It follows that context clauses for public units can only name other public units.
Partition library units. Partitions (like publics, packages and tasks) have specifications and bodies. Declaring a partition is equivalent to declaring an access type to an anonymous type. The following example illustrates how a partition is declared:

```
with ...;
partition Server is
  -- server interface;
end Server;
```

The notation is conceptually equivalent to:

```
with ...;
partition type Server_Type is
  -- server interface;
end Server_Type;
```

Here Server_Type is the name of the anonymous type of the Server objects. Because instances of partitions can only be referenced by access variables, the identifier Server_Type cannot be explicitly used in such references. The identifier Server_Type refers to the implicit access variable type that accesses instances of the partition. Where the type name is required, the new attribute 'Partition_Type is used with the name. In particular, this attribute can be used with the name Server to reference the partition type in an allocator. However, no partition object may be declared from the anonymous partition type.

Partition specification. The callable interface to a partition is defined by the specification of the associated partition library unit. Potentially this interface may be called by a partition on a remote processing site. Therefore, certain restrictions are imposed on the entities that can be defined. In particular, variables and types are not allowed. Variables are not allowed because their access would require a task in one node to access the local memory of another. Types are not allowed because two or more instances of the partition would imply that the visible types would be distinct between instances. This would lead to the need for dynamic type checking.

Task types are not allowed because the body of the task can access the internal state of the partition body. Consequently a remotely created task object would be able to access remote memory. A similar argument applies to generic units, and they are disallowed also. Partitions cannot be declared within a partition as a partition can only be a library unit.

Partitions may include any other library units in their context clauses except nodes (see section 4.2). To allow mutual references to partitions, an incomplete partition declaration (that is a partition name and any initialization parameters) can be placed into the library.

Partitions can declare initialization parameters in the specification part of their declaration. The types permitted are the same as those that can appear in the parameters to the visible procedures, functions and entries. Only in parameters are allowed. The example in section 4.3 will illustrate this.

Conformant partitions. In order to provide a mechanism for dynamic fault tolerance, we introduce the concept of conformant partitions to allow multiple partitions to have identical specifications and compatible access types. In particular, a partition can be declared to have the same specification as another partition as shown in the following example.

```
with ...;
partition Full_Function_Server is
  -- ...
end Full_Function_Server;
```

The above two partitions are called conformant partitions. They have the same type and therefore a variable declared to be an instance of Full_Function_Server can be used to reference a Degraded_Server. Conformant partitions are useful for programming degenerate modes, alternative implementation modes (e.g., simulations), and diverse programming techniques (e.g., N-version programming [12]).
4.2. Configuring a program

In order to provide a mechanism for managing the configuration of a program (its mapping onto logical processors) from within the program itself, we introduce another new kind of library unit, the node library unit. Analogous to partitions, the term *node* is also used to indicate both the node library unit and the transitive closure of the library units named in the node library unit context clauses. The purpose of a node is to collect together instances of partitions for execution on a single physical node in the target architecture. Restricting instances of partitions to be created within node library units maintains a clear separation between the configuring (and reconfiguring) components of the application and the partitions themselves. The node is recognised by the compiler/linker system as the unit which will be bound with an Ada run-time support system to form an executable binary load module.

Figure 6 illustrates the structure of a node. A node has the following format:

```ada
with ...;
node X is
  -- node interface
end X;

with ...;
node body X is
  -- code which defines the configuration
  -- and which reconfigures when necessary
begin
  -- any initialisation
end X;
```

In a similar manner to partitions these declarations implicitly declare an access type to an anonymous node type. Creation of an instance of a node requires the action of the allocator. Because instances of nodes can only be referenced by access variables the name X cannot be explicitly used in references to node instances. Rather, the name X refers to the implicit access variable type that accesses instances of the node. Where the type name is required, the new attribute 'Node_Type is used with the name. In particular, the attribute is used to reference access node types in an allocator. However, no node object may be declared from the anonymous node type. A node can name any other library unit in its context clauses (e.g., other nodes, partitions, publics, packages). As with partitions, packages shared between nodes are replicated in each node.

4.2.1. Partition creation within nodes. Partitions can be created within nodes (and therefore on the same processor as that on which the node is placed) by declaring a partition variable and then creating instances of the partition using an allocator. For example:

```ada
with Degraded_Server;
node body X is
  -- an uninitialised partition access variable
  Ds1 : Degraded_Server;
  -- an array type of 10 degraded servers access variables
  type Ads_T is array(1 .. 10) of Degraded_Server;
  -- an instance of a degraded server
  Ds2 : Degraded_Server := new Degraded_Server'Partition_Type;
  -- an array of 10 degraded servers
  Ads2 : Ads_T := (1 .. 10 => new Degraded_Server'Partition_Type);
begin
  -- the following creates an instance
  -- of a degraded server
  Ds1 := new Degraded_Server'Partition_Type;
end X;
```

A partition variable (that is an access variable indicating a partition instance) can be passed to another partition in the same node and to other nodes via the node interfaces.

4.2.2. Nodes and their creation. The restrictions that are placed on the interface to a node are exactly those that are placed on partition interfaces, and for basically the same reasons. Nodes can have initialization parameters, and conformant nodes can be declared. Any node can create another node using the allocator. An optimal parameter to the allocator indicates the logical processor on which the node is to be created. If no logical processor is specified an unallocated processor is chosen by the run-time support system. Only a single node can be created on each logical processor; multiple nodes per logical processor are not supported. One node, called the distinguished node, possibly designated as such by an appropriate pragma, is created automatically by the environment.

† Note that, where the network operating system provides the necessary capability, the actual node could be one of a number of virtual processing nodes in a single physical node.
4.2.3. System elaboration and termination. Separate load modules are created for each node type by the distributed Ada development environment. Each module is loaded onto one or more physical nodes in the target architecture (depending on how many instances of the node are created); as only one node per logical processor is supported, at most one load module is loaded on each logical processor.

System startup follows, but instead of a main procedure being called, the distinguished node is started by the underlying run-time support systems. The distinguished node elaborates according to Ada rules and commences execution. During elaboration or execution it creates instances of any local partitions. It may also create one or more nodes, indicating the processor on which each node should be created. The run-time support system for the distinguished node sends a request to the run-time support system on the target processor indicating that the node should be created and elaborated. The thread of control in the distinguished node which requested the creation is blocked until the new node has elaborated. However, parallel elaboration takes place if allocation is specified in a record or array aggregate. If creation or elaboration of the required node fails (either because the library does not contain a copy of the requested node, or because of an exception in the node's elaboration), then an exception (SInt_Error) is raised in the distinguished node. A successful elaboration returns the access pointer of the created node to the distinguished node. The elaboration and execution of a node may create further nodes and local partitions in a similar manner.

As partitions and nodes are library units, all tasks which they create will be library tasks. Note that no task hierarchies can be distributed across the network, although task hierarchies can exist within a partition. The distributed system will terminate when all library tasks are prepared to terminate. It is beyond the scope of this paper to discuss termination in detail, however the rules for termination are the same as those presented by Hutcheon and Wellings for York Distributed Ada [21].

4.3. An example AdaPT program

In order to illustrate some of the facilities of AdaPT, we outline a distributed program to implement the children's game of 'battleships.' Our goal is to illustrate how programs consisting of partitions and nodes are constructed, and to show the utility of conformant partitions.

Battleships is a game for two players. Each has a squared board representing a naval battle area, the squares being identified by the coordinates \((n, m)\) where \(n\) and \(m\) are integers. Each player arranges, at the start of the game, a certain number of ships on their board. A ship occupies one, two, three or four adjacent squares dependent on the kind of ship. (A battleship, for example, occupies four squares.) Details of the number of ships, the size of the board etc are not necessary for the purposes of this paper.

Having selected the disposition of their ships, these boards are fixed data. Players fire in turn at their opponents' boards. A move consists of selecting a square and announcing the choice to the opponent, who replies with the information 'missed', 'hit', or 'sunk'. The last situation arising when all the squares occupied by a ship have been hit. Each player now keeps a record of what has been discovered about the opponent's board by maintaining an image of that board, initially blank, on which information gained by the earlier shots are recorded. The winner is the first to sink all the opponents' ships.

In the simulation, players are modelled by partitions, each containing a task, which request inputs from and report the results to the console. It is convenient to introduce a referee, who keeps the boards with the fixed data of both players, receives the shots and reports on their effects. The complete program therefore consists of three partitions, two derived from a common type. It would be possible to configure the program on a network of one, two or three physical nodes. This logical arrangement is illustrated in figure 7.

A variant of the program would be a situation in which one of the players was a real person, while the other was the computer itself. In the second version the 'pseudo-player' would have the same interface as the real player who interacts via the keyboard and screen, but the implementation would be different. This will provide an example of a conformant partition.

Public. The following public unit provides the types for parameters of calls between the partitions.

```
public Shared_Type is
  -- This public unit provides the types to be shared
  -- between the partitions Player and Referee.

   type T_Turn is . . ;
   type T_Board is . . ;
```

![Figure 7. Arrangement of the players and the referee.](image-url)
type T_Move is . . ;
type T_Result is . . ;
end Shared_Types;

Partition. The following is the specification part of the partition Referee to represent the referee in a battleships game.

with Shared_Types;
use Shared_Types;
partition Referee is
  -- Referee keep copies of the boards of the two players.
  -- Each player has to initialise his/her board. Once this is done, Referee provides alternating access to the boards.
  -- In this way, each player plays a position on his/her opponent's board in turn until the game is over.
  -- To know when the game is finished, a unit can call Wait_for_End which will not return control until the game is over.
procedure Copy_Board (Owner : in T_Turn; Board : in T_Board);
function Play (Who : T_Turn; Move : T_Move) return T_Result;
procedure Wait_For_End;
end Referee:

The program would not be complete until the body defining the implementation is provided. However, as in Ada, this specification is enough to allow the definition of partitions which depend on the referee.

Next, the specification of the player partition (type) is shown:

with Referee, Shared_Types;
use Shared_Types;
partition Player (A_Referee : Referee; A_Turn : T_Turn) is
  -- Player simulates the behaviour of a player. After having its environment variables set up (through the initialisation parameters), it plays until the game is over.
end Player;

In the example, it would be natural to provide a package defining the data structure in which a player records its knowledge of its opponent's boards. Since that board will be examined in detail when choosing the next move, the package will provide also the algorithm for selecting the next move. There will, therefore, be a package Boards defined as follows:

with Shared_Types;
use Shared_Types;
package Boards is
  -- Declaration of the unique non-constant state item appearing in this package.
  -- Implementation of the services provided by the package.
  function User_Definition return T_Board is . . ;
end Boards;

This package, being 'withed' by the body of the player partition, will be replicated with each instance of the player, so each player will have its own copy of the board data structure.

A player is an active partition; it is implemented with a task in its body. It has a with clause for the Boards package, and declares an instance variable for the referee.

with Boards;
partition body Player (A_Referee : Referee; A_Turn : T_Turn) is
  The_Referee : Referee := A_Referee;
  My_Turn : T_Turn := A_Turn;

  task Game is
    My_Board : T_Board;
    Next_Move : T_Move;
    Result : T_Result;
    begin
      -- Initialisation of the two boards (the player's board and the view of the opponent's board).
      My_Board := Boards/User_Definition;
      Boards.Init_Opponent;
      -- Initialisation of the referee's copy of the player's board.
      The_Referee.Copy_Board (My_Turn, My_Board);
      -- Game cycle of the player.
      loop
        Next_Move := Boards/Choose_Move;
        Result := The_Referee.Play (My_Turn, Next_Move);
        exist when (Result = End_Of_Game);
        Boards.Update (Next_Move, Result);
      end loop;
      end Game;
    end Game;
end Player;

In this way, a partition provides the unit of distribution, aggregating closely-cooperating library units and exporting an interface to the rest of the program, so each partition can be written in familiar Ada style.

Nodes. Here, for simplicity, the whole program will be assembled for execution in a single physical node, so there is one node which constructs the complete program.

program node A_Single_Main is
pragma Distinguished;
end A_Single_Main;

with Referee, Player, Shared_Types;
use Shared_Types;
node body A_Single_Main is
  The_Referee : Referee := new Referee'Partition_Type;
  Player1 : Player := new Player'Partition_Type
    (The_Referee, Turn1);
  Player2 : Player := new Player'Partition_Type
    (The_Referee, Turn2);
begin
  -- The_Referee, Player1 and Player2 will run concurrently until
  -- the end of the game, synchronising to exchange
  -- moves.
  -- The node will not terminate until all library-level
  -- tasks in the three partitions have terminated.
  null;
end A_Single_Main;

The values Turn1 and Turn2 are literals of the type T_Turn.

Conformant partitions and conformant nodes. A set of conformant partitions allows different implementations of a partition to present a common interface (i.e., be of the same type). This provides a basis for managing mode changes and fault tolerance. The present example can be extended to allow a mode switch, in which the partition representing one of the players plays the game itself, instead of just providing an interface for a human player. Having defined a partition type in a partition declaration this can be used as the prototype for one or more conformant 'peers' with the identical interface specification by writing a declaration such as:

partition Automatic_Player
  (A_Referee : Referee; A_Turn : T_Turn) is Player;

This states that the external properties of the Automatic_Player are exactly those of the Player. However, the body might be entirely different. For example:

partition body Automatic_Player
  (A_Referee : Referee; A_Turn : T_Turn) is
  -- This player is the computer.
  ...
end Automatic_Player;

This alternative form of player might be selected at system construction if the human player wished to play against the computer rather than against another human. The following gives an outline of the code to construct a single node in this more general case.

with Referee, Player, Automatic_Player, Shared_Types,
  User_Interface;
use Shared_Types;
node body A_Single_Main is
  The_Referee : Referee; New_Referee : new Referee'Partition_Type;
  Player1 : Player := new Player'Partition_Type
    (The_Referee, Turn1);
  Player2 : Player := new Player'Partition_Type
    (The_Referee, Turn2);
procedure Init_Game is
  -- Initialises interactively the values of
  -- No_more_games, Turn_of_Player1 and Human_Players using
  -- User_Interface.

Supporting distribution and dynamic reconfiguration in AdaPT

function The_Other_Turn (A_Turn : T_Turn) return T_Turn is . . ;
  -- From a player's turn, it computes the turn
  -- corresponding to
  -- the other player.

task Life;
task body Life is
begin
  loop
    -- Tasks in the three partitions have terminated.
    The_Referee.Wait_For_End;
  end loop;
end Life;
end A_Single_Main;

4.4 Comparison with Ada9X

The above example has shown that it is relatively straightforward to express and configure distributed programs using AdaPT. It is instructive to consider briefly how this same example would have had to be programmed using the Ada9X constructs.

The public Shared_Types simply become a Pure package, and the AdaPT partition Referee becomes an Ada9X remote call interface package from which an active partition is created. Whereas the creation and configuration of the AdaPT Referee is specified within the program, enhancing the readability of the distributed program, the method of creating the active partition Referee and its configuration is unspecified by Ada9X.

More importantly, the AdaPT partition Player makes use of the fact that Player is a type. The actual instances are created under program control in the node A_Single_Main. Both players are created from the same partition type. This cannot be done in Ada9X. In Ada9X, Player1 and Player2 would have to be separately declared instances of a generic RCI package. While this might not be a major problem for a two player game such as battleships, more generally, it is a severe limitation.

Whereas configuration is unspecified in Ada9X, it is contained within the program in AdaPT, through the node A_Single_Main. While fault tolerance was not an issue in this example, the fact that configuration is included within the distributed program would make the management of fault tolerance much more readily accomplished in an understandable manner than through Ada9X. The mention of a conformant player partition provides a hint of how this might be accomplished.

Overall, not only is AdaPT a more flexible and
convenient language in which to express distribution and parallelism, but we have found it a powerful model against which other distributed and parallel languages can be compared, as a basis for evaluation.

5. Translating AdaPT to Ada83 and Ada9X

While we believe the AdaPT model to be more powerful than the Ada9X model, and to make the programming of distributed systems more readily accomplished, we do not have a compiler for it. It is therefore useful to consider how AdaPT might be translated into Ada83 or Ada9X. Such translators might then be implemented as preprocessors for one or the other versions of Ada. In this section, we consider these translations.

5.1. Translation of AdaPT to Ada83

In this section we briefly review how the above programs can be translated to Ada83. The actual code generated by the translation is given in full by Goldsack et al [19].

5.1.1. Public units. Public units present no problem for transformation; they are replaced by normal Ada packages, restricted only in that they possess no variable state.

5.1.2. Partitions. The partitions and nodes are structures with features similar to those which package types [25] might have if they existed in the language. The basis of the translation which we have developed into Ada83 depends crucially on the fact that they are types.

Partitions in AdaPT are types whose instances can be thought of as abstract state machines (ASMs). They possess persistent state attributes encapsulated within their bodies, while presenting in their interfaces sets of operations which can modify their states. They may be active or passive. In the first case they have one or more internal tasks which can cause changes of state to occur due to the partition’s own actions. If the partition is passive, however, its state changes only as a result of invocation of the subprograms in the interface.

To create a program in Ada83 whose behaviour is equivalent to the effect of an AdaPT partition we have shown (informally) that for every partition type there is an abstract data type (ADT) whose instances have the equivalent effects to those of the partition instances. The full account of the transformation will not be given here [19]. Essentially it implies the collection of all the state elements of the partition (and all its tasks if it is active) into a state record. This record is now a data type which is exported as a private access type. Client program units can declare instances of this access type as they would have declared instances of the partition. The procedures and functions declared in the partition unit are modified to take an extra parameter of the access type, which is passed with the data in every call.

In AdaPT a partition unit can ‘with’ other library units. In our translated version of such a complex partition, these are also translated to abstract data types defining state records. For each such ‘withed’ package, an instance of that record is declared within the state record of the partition which has the relevant with clause among its context clauses. Thus the original structure of the partition as a tree† of ‘withed’ units is preserved in the translation.

Here, for example, is the translation of the Player partition. This exports a private access type, and the definition of the state record can be deferred to the body. Since the private type cannot be instantiated by a client with the new allocator, it is necessary to introduce a Create operation to provide for its effect. It is important to note that, even if the access type is not private, there is a difference between executing a new allocator in the client, and in the body of a Create operation which is part of the ADT. This concerns the location of the stored record in a distributed system. In the latter case it is correctly stored on the heap of the unit executing the operations. For symmetry there is again also a Destroy operation.

with Referee_Adt, Shared_Types;
use Shared_Types_Adt;
package Player_Adt is
   -- Player simulates the behaviour of a player. After
   -- having
   -- its environment variables set up (through the
   -- initialisation
   -- parameters), it plays until the game is over.
   type Player is private;
      -- Create simulates the operator new for the partition
      -- creation and supports the initialisation parameters.
   procedure Create ((P : in out Player);
   A_Referee : in Referee_Adt.Referee;
   A_Turn : in T_Turn);
      -- Destroy provides the complementary operation to
      -- Create.
   procedure Destroy (P : in out Player);
   private
      type Player_State;
      type Player is access Player_State;
   end Player_Adt;
with Boards_Adt;
package body Player_Adt is
   task type Life_Type is
      -- An extra entry is added to give this task access
      -- to the state of a Player's instance. The entry will
      -- be called by the CREATE operation.
   entry Set_Init (P : in Player);
   end Life_Type;

   type Player_State is
      record
         -- An instance of theBOARDS_ADT's state.
         B : Boards_Adt.Boards;
         -- State variables derived from the body of Player.
         The_Referee : Referee_Adt.Referee;
         My_Turn : T_Turn;
         T : Life_Type;
      end record;
      -- Note how the state record of the player is now
      -- composed with that of the BOARDS package.
   end Player_Adt;

† The general form of the dependency graph is an acyclic directed graph. This presents difficulties in the translation. A preliminary program transformation to a tree structure.
The result is a program which is quite well structured in Ada83 terms. ADTs have been much discussed as Ada structuring features. To write such an ADT directly is quite feasible. However, the ADT is not as clear as the AdaPT partition itself, and since ADTs can be written which are more general than the partition, a programmer must observe some restrictions in writing them. Nevertheless, the use of a complex data type as a partition (or virtual node) in a distributed system is an option which is available in Ada83 and has been largely overlooked by workers (including ourselves) who have previously sought appropriate ways of writing virtual nodes and composing distributable Ada programs.

We consider, however, that the construction from first principles of a neatly structured ADT to represent a large partition is not easy, and the direct definition of the large data structures involved may not always seem natural to the programmer. One solution might be to retain AdaPT as a methodology, producing the translation by hand or using a pre-processor. In a later section we shall see, however, that in Ada9X, the use of class-wide types will provide a natural way to develop partitions for distribution.

5.1.3. Nodes. AdaPT introduces a separate concept of a node whose destiny is to become the source code representation of a binary executable unit to run on a processor in the network. A node differs little in its structure from a partition and it too can be converted to an ADT.

5.1.4. Conformant partitions and nodes. Conformance is a type of polymorphism not supported in Ada83. Objects of the same type are differently implemented. Since an access variable can only be bound to objects of one type, it is necessary to use the type conversion facility offered by the generic function Unchecked_Conversion to achieve the effect. It is however possible to do so in a controlled way, hidden in a procedure body.

5.2. Translation of AdaPT to AdaPX

5.2.1. Public units. As in the case of Ada83, publics are presented in Ada9X as packages which have no variable state. Chapter 10 of the Ada9X Reference Manual [1] defines a pragma Pure to label a package which has this property. It will not change the meaning of the program, but will make possible the construction of a tool to check that the package has the intended property, and also to check at compile time that attempts are not made to share packages not so labelled.

5.2.2. Partitions. There is little doubt the Ada9X object-oriented style of programming will provide a useful way of developing the kind of structures required for the ADTs that we use as partitions. It is of some interest that those features which are inherited by a class-wide type hierarchy are precisely the features (the type and its operations) which form an ADT.

As noted in the section on partitions, the 'withed' package associated with a partition closure, which is replicated with each instance of the partition, has an effect equivalent to type extension in a derived type.

To keep the account of the Ada9X form of the translation as simple as possible, we present first a purely schematic outline of a partition in which a partition unit C 'withs' a package B which in turn 'withs' a package A. Consider first the packages on which the partition depends. They consist of the package B which 'withs' package A:

```plaintext
package A is
  procedure Pa;
end A;
with A;
package B is
  procedure Pb;
end B;
```

In Ada9X, the aggregation of the states can receive language support by the use of record extension. We make ADTs corresponding to the separate packages A and B and then form the aggregated state by type derivation. To make such extensions legal, the type to be extended must be declared to be a 'tagged' type [1].

```plaintext
package A_Adt is
  type A_State is . . . ;
  procedure Pa (Va : in out A_State);
end A_Adt;
package B_Adt is
  type B_State is tagged private;
  procedure Pb (Vb : in out B_State);
private
  type B_State is . . . ;
private
  type B_State is . . . ;
end B_Adt;
with A_Adt, B_Adt;
use A_Adt, B_Adt;
package Full_B_Adt is
  type Full_B_State is new B_State with private;
  procedure Pb (Vb : in out Full_B_State);
private
  type Full_B_State is new B_State with record
    Va : A_State;
  end record;
  -- This defines a state composed by B_State and
  -- A_State.
end Full_B_Adt;
```

Note that it is the type exported by package B which is inherited and extended; package A provides a state type that extends the derived type. The operations such as Pa, exported by package A are not available as operations callable by users of the derived type; however, nor were the operations exported by the package A available to users of B when B was a package. In both cases these operations were accessible for calling from the bodies defining the operations in the interface.

Next we must form the partition. Here the process is exactly the same, except that the partition exports an access type to provide the type representing the partition. The partition C is first converted to an ADT on its own, and
then the full partition is created by inheriting C and extending its type by aggregating its state with an instance of B. We should note, however, that the procedures created in the partition have variables of the state record type as arguments. It is a feature of the new Ada that such subprograms can be called with actual parameters of access type, provided they are given the new mode access in the formal part.

```
package C_Adv is
private
end C_Adv;
```

```
with B_Adv, C_Adv;
use B_Adv, C_Adv;
package Full-C_Adv is
  type C_State is tagged private;
  procedure Pc (Vc : in out C_State);
  type C_State is tagger private;
  type Full-C_State is new C_State with private;
  type Full-Ptr is access Full-C_State;
  procedure Pc (Vc : in Full-C-Ptr);
  procedure Create (Vc : in out Full-C-Ptr);
  procedure Destroy (Vc : in out Full-C-Ptr);
  type Full-C_State is new C_State with private;
  record
    Vb : Full-B-Ptr;
  end record;
end Full-C_Adv;
```

### 5.3. Conformant partitions

In section 5.2.4 we drew attention to the difficulty of constructing conformant partitions in Ada83 in view of the strong typing rules of the language Ada9X supports the notion of tagged types [35] which can be utilized to make provision for conformance. First we continue the schematic forms used in the previous section to describe a prototypical partition B and a conformant partition Cb. In AdaPT these would be specified:

```
partition B is
  procedure Pb;
end B;
partition Cb is B;
```

In the following we present a possible Ada9X program for the same purpose; it entails defining an empty (fully abstract) tagged type, and deriving two different implementations from it. The following is a possible outline.

First we have a fully abstract definition of a type Empty_State with an operation over it, Pb. There is no body to this package, since neither object is further defined. The package exports an access type for referencing instances of the tagged state record. Since this is defined to be a class pointer, it is permitted to reference all descendents of Empty_State in the derivation hierarchy. Tagged types carry a 'tag' which permits run-time recognition of the current variant.†

```
package Abstract_B is
  type Empty_State is abstract tagged private;
  type Class_Ptr is access all Empty-State'Class;
  type Empty-State is abstract tagged null record;
end Abstract-B;
```

This is followed by two alternative packages, inheriting the same tagged type and extending it in each of two different ways.

```
with Abstract_B;
package B_Adv is
  type B_State is new Abstract_B.Empty_State with private;
  type B-Ptr is access all B_State;
  procedure Pb (Vb : access B_State);
  function Create return B-Ptr;
  procedure Destroy (Vb : in out B-Ptr);
  private
  type B_State is new Abstract_B.Empty_State with record
    -- State required by the implementation of B.
end record;
end B_Adv;
```

```
package Cb_Adv is
  type Cb-State is new Abstract_B.Empty-State with private:
  type Cb-Ptr is access all Cb-State;
  procedure Pb (Vb : access Cb-State);
  function Create return Cb-Ptr;
  procedure Destroy (Vb : in out Cb-Ptr);
  type Cb-State is new Abstract_B.Empty-State with
  record
    -- State required by the implementation of CB.
end record;
end Cb_Adv;
```

The following is a fragment of code showing how this type would be used by a client.

```
Vb : Abstract_B.Class_Ptr;
begin
  Vb := B_Adv.Create;
  Abstract_B_Pb (Vb); -- A call to B_Adv.Pb.
  Vb := Cb_Adv.Create;
  Abstract_B_Pb (Vb); -- A call to Cb_Adv.Pb.
end;
```

### 5.3.1. The node. Having constructed types whose instances constitute the virtual nodes of a distributable program in Ada9X, they can now be assembled into one or more 'supertypes' whose instances form the nodes of the program. The treatment is exactly like that of the partition: the node is presented as an ADT exporting an access type, and whose state is a record containing instance variables for each of the component partitions. These are instantiated

† Note that the AdaPT conformant partition has a problem in that the overhead of providing for possible conformance is carried by all partitions, because there is no syntactic recognition of those partitions which may have conformant peers. Limiting the polymorphism to tagged types avoids this difficulty in Ada9X.
by the Create operation of the node, calling the respective 
create operations of the partitions. Destroy works in a 
similar way.

5.3.2. Forming a distributed system. Within an 
application, a node is to become the code from which a 
binary load module can be generated for allocation to a 
particular physical node in the network. When there are 
several nodes, to become binaries for allocation on 
different physical node, the transformation into Ada 
introduces the following new problems:

- system-wide identifier,
- node creation,
- remote communication.

System-wide identifier. In AdaPT, partitions and nodes 
define access types. To call an operation exported by a 
partition or a node instance the caller must use a reference 
to the instance.

R : Referee := new Referee'Partition_Type;
... 
begin
  R.Copy.Board (A_Player, A.Board);
... 
end;

Until now, in the transformation of partitions and nodes into 
Ada9X (as well as into Ada83), this reference has been 
implemented using an Ada access type. As partition and 
node instances may be located for execution on different 
nodes of a network, the use of an Ada access type to refer to 
these partition or node instances is sufficient. In a 
networking environment, an Ada access object only makes 
sense if it is related to the machine whose storage space it 
addresses. Therefore, in the transformation of partitions and 
nodes into Ada9X, the ADT's type must be extended adding 
a node identifier (i.e. program and machine identifiers).

Node creation. AdaPT differentiates two kinds of nodes, 
distinguished and non-distinguished. In any AdaPT's 
system, the designer must define a distinguished node 
which will be the starting point for the system's 
 elaboration. This node is started by the operating system 
when a user runs the system. To get the same effect in 
Ada9X, we need to declare a procedure as a main program 
for the distinguished node.

with Nodel_Adt;
procedure Main_Nodel is 
  N1 : Nodel_Adt.Node;
begin
  Nodel_Adt.Create (N1);
end Main_Nodel;

The non-distinguished nodes are remotely created by other 
 nodes. In the AdaPT creation statement the location for the 
new node can be specified as a parameter to the allocator, as 
shown below in an example in which Nodel creates Node2.

N2 : Node2 := new Node2'Node_Type 
  (A_Network_Location) (R);

A non-existent location must raise the predefined AdaPT 
exception Site_Error. The creator node has to wait until the 
creation is finished (i.e. when a reference to the created 
ode is returned).

In terms of our translation, the creation of a node 
instance can be understood as a function call which returns 
a reference to the node instance (state record) created. 
Therefore, we could use an Ada function on the proper 
target machine to implement the main program for Node2.
A simplified version of such function is shown below.

with Node2_Adt, Referee_Adt;
function Main_Node2 (R : Referee_Adt.Refererce) return 
  Node2_Adt.Node2 is 
  N2 : Node2_Adt.Node2;
begin
  Node2_Adt.Create (N2, R);
  return N2;
end Main_Node2;

Since N2 would, in general, contain a task instance, this 
task instance would continue to run until the task 
terminates, even after main function returns the reference 
and finishes, thus implementing Node2. In practice, to 
make the programs portable, a more complex solution is 
required due to constraints imposed by UNIX (e.g. a 
command cannot return a result until its execution is 
finished), by Ada compilers (e.g. full support of functions 
as main programs) and by the possible raising of an 
exception that must be propagated over the network.

As for the creator node (e.g. Node1), the creation of a 
non-distinguished node in Ada9X involves the following 
steps:

- to check whether the location specified is correct, raising 
  the exception Site_Error in the last case,
- to fetch the code corresponding to the Ada program 
generated from the definition in AdaPT of the node (e.g. 
Main_Node2 from Node2),
- to copy and to start this code on the specified network 
  node (i.e. A_Network_Location),
- to wait for the result of the creation (i.e. the reference to 
  the node instance created or an exception).

Remote communication. In itself the implementation of 
remote communication for Ada programs is not a new 
topic. The issues associated with remote procedure calls 
(RPC) and remote entry calls (REC) have been already 
widely studied in many projects. In our translation of 
AdaPT into Ada9X, we have simply reused the main 
ideas of [3] which define a 'source-level' approach; the 
code for supporting the remote communication is 
introduced by a transformation tool. In this approach, a 
distinction is established between a transport layer which 
provides a standard communication interface and a remote 
rendezvous layer which builds the RPC and REC on top of 
the transport. Roughly speaking, the rendezvous layer 
corresponds to the ISO's session and presentation layers.
5.3.3. Distribution. In the battleships example, the three partitions were composed into a single program by constructing a node with instances of each, and calling the appropriate procedures to interconnect them. This node will then be used to generate an executable binary.

It is equally possible to construct a distributed program by forming two or more nodes, and configuring the partitions appropriately between them. These then form a collection of executables for execution of different nodes in the network.

6. Conclusion

The new Ada9X standard will provide basic support for partitioning applications for execution in a distributed environment. However, it falls short of providing the full expressive power that would be expected of a language specially designed to support distributed system. In particular, the unit of distribution is not a first class language object and methods for dynamic configuration are left to the implementor.

In this paper we have proposed alternative extensions to Ada to support partitioning of distributed applications. An AdaPT program is considered to consist of two basic subsystems: one for detailing the functionality of the application components, the other for addressing system-wide issues such as program configuration and reconfiguration. We have introduced three principal new constructs: publics to supply types to be shared across machines, partitions to be the unit of distribution, and nodes to permit the dynamical configurations of distributed programs. Partitions and nodes are first class language objects, and provide some of the flexibility missing in Ada9X. It is obvious that there is a great deal of similarity in the capabilities of nodes and partitions. We have considered the possibility of merging the concepts. This has a number of potential advantages, not least of which is reducing the number of new concepts in the proposal. The prime reason we have resisted this temptation is that it would reduce the clear separation between concerns for functional and configuration aspects of the application.

We have been pleased to see some of the key issues addressed by Ada9X. In particular:

- **Partitions**
  The notion of a partition is supported in Ada9X but the mechanism by which library units comprise a partition is left as part of the implementation. Pragmas allow the programmer to identify which library units can be used as interfaces between partitions.

  In AdaPT, a partition is identified by the interface it provides and is supported by language syntax.

- **Shared data types**
  In Ada9X shared data types are collected together in packages which are identified by a pragmas (the pure pragmas). In AdaPT, shared types are collected together in special library units called public library units.

- **Configuration**
  The configuration of an Ada9X program is implementation dependent.

  The configuration of an AdaPT program can be specified as part of the program in an implementation independent manner.

- **Reconfiguration**
  In Ada9X a limited form of reconfiguration is supported by the use of pointers to subprograms and class-wide programming using tagged types.

  AdaPT allows the programmer to write reconfiguration algorithms in the language.

In summary, Ada9X removes some of the limitations of Ada83 in the area of programming distributed systems. However, its solutions are conservative and do not provide the expressive power and flexibility that can be obtained with more radical changes.

**Acknowledgments**

The ideas expressed in this paper are based upon those developed at the Third International Real-Time Ada Issues Workshop by the Distributed Systems/Virtual Nodes Working Group. The authors would like to acknowledge Kent Power for his help in developing some of the initial concepts. We would also like to thank Offer Pazy for his help in developing some of the mappings to Ada9X.

This work has been sponsored, in part, by NASA subcontract 4074 Cooperative agreement NCC-9-16.

**References**


