NASA-CR-193117

Transforming AdaPT to Ada9x

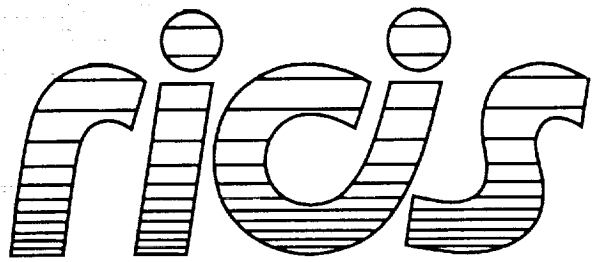
Stephen J. Goldsack
A. A. Holzbach-Valero
Imperial College, London, England

Richard A. Volz
Raymond S. Waldrop
Texas A&M University

April 8, 1993

Cooperative Agreement NCC 9-16
Research Activity No. SE.35

NASA Johnson Space Center
Engineering Directorate
Flight Data Systems Division



Research Institute for Computing and Information Systems
University of Houston-Clear Lake

TECHNICAL REPORT

RICIS Preface

This research was conducted under auspices of the Research Institute for Computing and Information Systems by Raymond S. Waldrop and Richard A. Volz of Texas A&M University and A. A. Holzbacher-Valero and Stephen J. Goldsack of Imperial College, London, England. Dr. E.T. Dickerson served as RICIS research coordinator.

Funding was provided by the Engineering Directorate, NASA/JSC through Cooperative Agreement NCC 9-16 between the NASA Johnson Space Center and the University of Houston-Clear Lake. The NASA research coordinator for this activity was Terry D. Humphrey of the Systems Software Section, Flight Data Systems Division, Engineering Directorate, NASA/JSC.

Transforming AdaPT to Ada9x

Task T8 Report

NASA Subcontract #074
Cooperative Agreement NCC-9-16
Research Activity # SE.35

Period of Performance: May 1, 1990 - March 31, 1993

Submitted to
RICIS

Submitted by
S. J. Goldsack, Imperial College, London, England
A. A. Holzbacher-Valero, Imperial College, London, England
R. Volz, Texas A&M University
R. Waldrop, Texas A&M University

Transforming AdaPT to Ada 9X

S.J.Goldsack, A. A. Holzbacher-Valero,
Imperial College, London
R.Volz, R.Waldrop
Texas A&M University

April 8, 1993

Abstract

This paper constitutes report R8 for NASA subcontract #074 Cooperative Agreement NCC-9-16. It describes how the concepts of AdaPT can be transformed into programs using the object oriented features proposed in the preliminary mapping for Ada9x. Emphasising, as they do, the importance of data types as units of program, these features match well with the development of partitions as translations into Abstract Data Types which was exploited in the Ada83 translation covered in report R3. By providing a form of polymorphic type, the Ada9x version also gives support for the conformant partition idea which could be achieved in Ada83 only by using *UNCHECKED_CONVERSION*.

The paper assumes that the reader understands AdaPT itself, but briefly reviews the translation into Ada83, by applying it to a small example. This is then used to show how the same translation would be achieved in the 9x version.

It is important to appreciate that this paper does not use, or even discuss in any detail, the distribution features which are proposed in current mapping, as those are not well matched to the AdaPT approach. Critical evaluation and comparison of these approaches is given in a separate report.

1 Introduction

In report R3 of the present project, a study was presented of the translation of an AdaPT program into executable code in Ada83. Since that report further work has been done on this subject, which has been presented in an up-dated version of that report.

An original aim of the project was to develop further the AdaPT proposal with the intention of influencing, or at least helping to guide the mapping team in developing an approach to distribution for Ada9x. Since that time, the mapping team has issued their language proposals, including an annex which supports the distribution of Ada 9X programs. Important changes proposed in Ada which affect this project are:

- support for a form of inheritance through type extension, based on an adaptation of the Ada83 concept of the derived type, and
- the introduction of a model for distribution through which Ada programs can be developed in a standard way for distributed targets.

The present report describes the extension of the work in T3 to cover Ada9x. To a large extent the changes to Ada are upwards compatible with Ada83, so programs developed using our translation of AdaPT into Ada83 will also provide correct Ada9x programs. However, in several ways the proposals for the mapping give enhanced support for AdaPT, and simplify the translation.

- Record aggregation, as used to translate partitions with "withed" packages into Ada83, is more directly provided by the use of type derivation in the proposed new version of the language.
- A form of polymorphism, permitting execution time binding of procedure calls to alternative versions of the procedure definition enables conformant partitions to be more directly supported, without the use of unchecked conversion.

The Ada83 translation makes extensive use of the notion of an abstract data type to translate the partition. The importance of such structures is greatly enhanced in Ada9x, since the model for object oriented structuring using inheritance is mainly developed around the data type. It is therefore straightforward to relate our Ada83 translation approach to one for the proposed form of Ada9x. The present paper aims to show how this works, and how the AdaPT notions can be supported in the proposed form of Ada9x.

The distribution features of the new language mapping also turn around a concept of a partition. This is not, however, the same as the partitions in AdaPT, and they are not used in the present translation. For a comparison of the two approaches to distribution, please see report R2R which makes a detailed comparison of the approaches, and provide a critique of the mapping proposal.

2 The Example

Throughout this paper, the features of AdaPT and their translations into Ada83 and Ada9x will be illustrated by outlining a program to simulate the children's game of "battleships". This 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 his 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 now 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 he has discovered about his opponent's board by maintaining an image of that board, initially blank, on which he gradually fills in information gained by his earlier shots. The winner is the first to sink all his opponent's ships.

In the simulation, players are modelled by partitions 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 virtual nodes, two derived from a common type. It would be possible to configure the program on a network of one two or three physical processors. This arrangement is illustrated in figure 1.

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.

2.1 Public

As explained above, a public unit allows the sharing of definitions of types, constants and services between nodes or between partitions within nodes. To this end, it has the following features: it is identical to a package, except that it is not allowed to have variable state. When appropriate, the defined types may be private. A public unit may introduce type definitions with associated operations, forming an Abstract Data Type. It may have **with** clauses for other public units, but for no packages.

In the example the following public unit provides the types for parameters of calls between the partitions.

```
public Shared_Types is
```

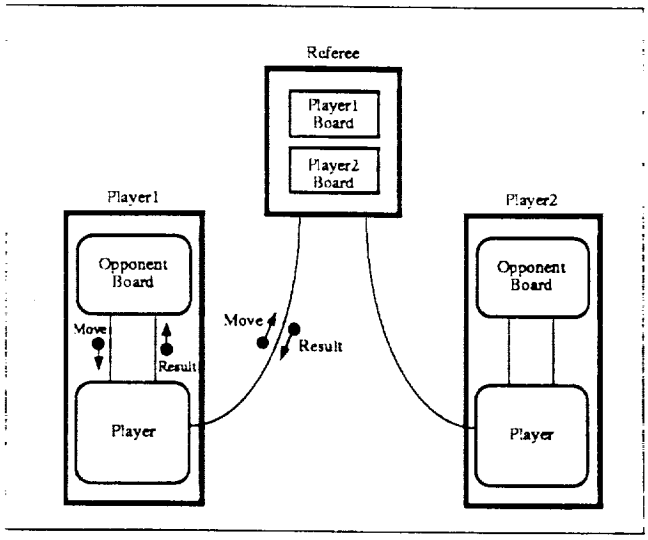


Figure 1: arrangement of the players and referee.

```
      -- This public unit provides the types to be shared between the
      -- partitions Player and Referee.
   type T_Turn is ..
   type T_Board is ..
   type T_Move is ..
   type T_Result is ..
end Shared_Types ;
```

2.2 Partition

A partition is a new library unit capturing the concept of a virtual node type and whose instances have the properties required of virtual nodes. One can recognise in the partition a restricted form of *class* as found in object oriented languages.

A partition declaration defines a type which structurally speaking is similar to a package. It can be noted that a partition is just like a package in Ada except for certain constraints. Like a package, it has a *specification* part and a *body*. Its specification defines the interface to the partition instances in terms of subprograms and tasks which are made available for use by its clients, while its body encapsulates its implementation. A partition may have an active "thread of control", in which case it contains one or more tasks. Unlike a package no type or object declarations are allowed in a partition specification to avoid remote memory accesses and dynamic type checking.

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 keeps 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 standard Ada, this specification is enough to allow the definition of partitions which depend on the referee.

This is followed here by the specification of a player:

```
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 thinks and plays until the game is over.

end Player ;
```

The name associated with a partition declaration does not represent, as would be the case in a package declaration, a single instance of the object with this definition in the library. It is the name of an access type bound to an anonymous partition type.

The declaration:

```
with Referee ,
   Shared_Types ;
use Shared_Types ;
partition Player (A_Referee : Referee ; A_Turn : T_Turn) is ...
```

can be understood to mean:

```
with Referee ,
    Shared_Types ;
use Shared_Types ;
partition type Player_Type (A_Referee : Referee ; A_Turn : T_Turn) is ...

      -- the declaration of an anonymous package type
type Player is access Player_Type ;
```

Partition variables may be declared in the declarative parts of partitions or nodes. Such a variable is an access variable and is initially **null**. A partition instance is created by a **new** allocator, as is usual for an object referenced by an access type in Ada.

```
A_Player : Player := new Player'PARTITION (The_Referee , The_Turn) ;
```

Note that the syntax is exactly that of the **new** allocator in Ada83; however, since the type of the constructed object must be given, the attribute 'PARTITION was introduced to give the hidden type of the player objects. The brackets following the keyword PARTITION contain the actual parameters matching the formal parameters defined in the declaration of the partition, and giving the initial configuration of the system.

This allocator may not be executed by a partition. Partition variables in a partition can receive values only by assignment from instance variables which are created at the node level. Preventing the creation of partitions by partitions in this way guarantees that the new units are not misused as units for software development displacing the package concept as the main modularity feature in a normal Ada program.

A programmer is encouraged to use normal Ada style in developing a partition. A partition or its body may have a context clause "withing" one or more normal Ada packages. If these packages have "state", then they are replicated with each instance of the partition which "withs" them. If these in turn "with" further packages, then those too are replicated. The complete dependency graph of the partition up to but not including any other partitions which may be "withed", is replicated with each instance. The partition unit with this set of packages on which it depends is sometimes referred to as the closure of the dependency graph. In fact, the word *partition* is overloaded, being used informally to mean the partition closure, the set of all the units appearing in its dependency graph except other "withed" partitions while serving also as the reserved word identifying the interface unit.

In the example, it would be natural to provide a package defining the data structure in which a player records his knowledge of the opponent's board. 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

      -- This package provides to the player:
      -- the definition of his/her board by the user,
      -- operations to manage a view of his/her opponent's board.

      -- Definition by the user (the player) of his/her ships'
      -- arrangement in the player's board.
   function User_Definition return T_Board ;
      -- Operations based on the player's view of his/her opponent board.
   procedure Init_Opponent ;
   procedure Update (Tried_Move : in T_Move ; Result : in T_Result) ;
   function Choose_Move return T_Move ;
end Boards ;



package body Boards is

      -- Declaration of the unique non-constant state item appearing
      -- in this package.
   Opponent_Board : ..

      -- Implementation of the services provided by the package.
   function User_Definition return T_Board is
      .
      .
      .
end Boards ;
```

As explained above, this package, being "withed" by the body of the player partition, will be replicated with each instance of the player, so each will have his 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 Life ;
   task body Life 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) ;
         -- Life cycle of the player.
```

```
        loop
            Next_Move := Boards.Choose_Move ;
            Result := The_Referee.Play (My_Turn , Next_Move) ;
            exit when (Result = End_of_Game) ;
            Boards.Update (Next_Move , Result) ;
        end loop;
    end Life ;
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. Finally, we note again that since partitions are units of distribution, the system designer must be conscious that inter-partition communication is potentially much slower than intra-partition communication.

2.3 Nodes

Nodes are intended to provide the system designer or programmer with control over configuration, the way in which partitions are allocated in the network. Structurally speaking, they are very similar to partitions. A node possesses a specification which provides the node interface and has to fulfill the same constraints as a partition specification. It also has a body that implements the services offered by the node. As with the partition, the declaration of a node introduces an access type name bound to an anonymous type. Node variables can only appear in the declarative part of a node. Node instances can be created by other nodes through the execution of the **new** allocator. Unlike a partition, a node can create other node instances as well partition instances. If this distinction were not made, then a programmer could develop a hierarchy of partitions, violating the intention of their introduction.

There is a one to one mapping between node instances and executable binary modules. Communication between node instances is always via message passing. One node, identified by a pragma *distinguished*, is selected by the programmer as a "distinguished" node. This node is logically the main program and is created, elaborated and started by the operating system at system initialisation. Others are created and started by other nodes.

In the design of AdaPT it was felt that, despite their structural similarity to partitions, the purpose of the node, aggregating elements of a program which will form a single binary, was sufficiently different from that of the partition that it was sensible to introduce it as an independent concept.

Here, for simplicity, the whole program will be assembled for execution in a single machine, so there is one node which constructs the complete 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 ;
     Player1  : Player  := new Player PARTITION (The_Referee , Turn1) ;
     Player2  : Player  := new Player PARTITION (The_Referee , Turn2) ;
begin
       -- The_Referee, Player1 and Player2 will run concurrently until
       -- the end of the game.  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 type *T_turn*.

2.4 Conformant Partitions and Conformant Nodes

In the design of the AdaPT extensions of Ada, it was felt that the needs of reconfiguration and error recovery frequently require a switch of mode, in which a set of operations is replaced by others with the same interface, but differently implemented. Thus, what was required was the ability to define partitions of the same logical type, but differently implemented. A mode change could then be obtained by assigning a different value to some instance variable in the caller. This is a form of polymorphism not available in Ada83. It is provided in AdaPT by a concept of conformant partitions.

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. We do not propose to expand on this idea here, but the 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;
   No_more_games,
   Human_Players : BOOLEAN;
   Turn_of_Player1 : T_Turn;
   Player1,
   Player2 : Player;
   procedure Init_Game is..
      -- Initialises interactively the values of No_more_games,
      -- Turn_of_Player1 and Human_Players using User_Interface.
   procedure 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
         Init_Game;
         exit when No_more_games;
         The_Referee := new Referee'PARTITION;
            -- We assume that the first player is always human.
         Player1 := new Player'PARTITION (The_Referee, Turn_of_Player1);
         if Human_Players then
               -- The two players are human.
            Player2 := new Player'PARTITION
                  (The_Referee, The_Other_Turn (Turn_of_Player1));
         else
               -- The second player is the computer.
            Player2 := new Automatic_Player'PARTITION
                  (The_Referee, The_Other_Turn (Turn_of_Player1));
         end if;
         The_Referee.Wait_for_End;
      end loop;
   end Life;
end A_Single_Main;
```

3 Translation of AdaPT to Ada83

As most aspects of our suggested mapping between AdaPT and Ada9x have their roots in the transformation of AdaPT into Ada83, it is useful to review the key ideas of this latter transformation. This section describes how this translation can be formulated for each of the AdaPT units. A more detailed discussion of this translation process may be found in the revised report for task T3 of this project.

3.1 The public: a shared repository of definitions.

Publics present no problem for transformation: they are replaced by normal Ada packages, restricted only in that they possess no variable state.

3.2 Partitions

The partitions and nodes are structures with features similar to those which package types 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. 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 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[1] of "withed" units is preserved in the translation[2].

Here, for example, is the translation of the complete Player partition including the *Boards* package which in this case forms the whole of the dependency graph.

```
with Shared_Types ;
use Shared_Types ;
package Boards_ADT is

      -- This package provides the player with an instance of the board type:
      -- In this he maintains a record of all he has learned of his
      -- opponents' ship arrangement.  This package also provides the
      -- operation to construct the player's own battle arrangement
      -- to send to the referee.  Also the operations to actually
      -- conduct the game by selecting a move and recording the result.

   type Boards is private;
      -- Definition by the user (the player) of his/her ships'
      -- arrangement in the player's board.
   procedure User_Definition (B : in out Boards ; The_Board : out T_Board) ;
      -- Operations based on the player's view of his/her opponent board.
   procedure Init_Opponent (B : in out Boards) ;
   procedure Update (B : in out Boards ; Tried_Move : in T_Move ; Result : in T_Result) ;
   procedure Choose_Move (B : in out Boards ; The_Move : out T_Move) ;
private
   type Boards is
      record
         Opponent_Board : ... -- as defined in the body of Boards.
      end record ;
      -- This is the state record for board instances.
end Boards_ADT ;
```

There will be additionally a *Create* operation if there is an initialisation part in the ASM and a *Destroy* operation if there is a task in the state record.

We come now to the translation of the partition unit itself. 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 owning the operations. For symmetry there is again also a *Destroy* operation.

```
with Referee_ADT ,
```

[1] The general form of the dependency graph is an acyclic directed graph. This gives difficulties in the translation. Such a graph can be modified by a preliminary program transformation to a tree structure. We do not consider these details here.

[2] We shall see later, when considering the type extensions supported in Ada9x how this feature can be interpreted as inheritance with type extension.

```
   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 thinks and 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_Initial (P  : in Player );
   end Life_Type ;

   type Player_State is
      record
           -- An instance of the Boards_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 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 result is not as neat 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 it. 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 derived types will provide a natural way to develop partitions for distribution.

3.3 Nodes As Configuring Units

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 machine in the network. A node differs little in its structure from a partition and it too can be converted to an ADT. In our translation, however, it is supplied with a main procedure which creates and elaborates the node instance when it is invoked during system start-up by the operating system by a command from the user if it is the distinguished node, or by another node if it is not[3].

3.4 Conformant Partitions and Nodes.

Conformance is a type of polymorphism not supported in Ada 83. 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. The technique was explained in the T3 report, and we do not propose to describe it in detail here.

4 AdaPT and Ada9X

4.1 Public Units

As in the case of Ada83, publics are presented in Ada9x as packages which having no variable state. The mapping proposes to introduce a pragma PURE to label a package which has these properties. 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. The public unit given in section 2.1 will therefore be rendered:

```
package Shared_Types is
pragma PURE;
```

[3] An account of the work on partition to partition and node to node communications is given in the extended T3 report

```
      -- This public unit provides the types to be shared between the
      -- partitions Player and Referee.
   type T_Turn is ..
   type T_Board is ..
   type T_Move is ..
   type T_Result is ..
end Shared_Types ;
```

4.2 Partitions

The reports of the Ada9x mapping team describe at some length their ideas for developing an object oriented style of Ada programming by extending the notion of the derived type. There is little doubt that such features 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 of a package which are inherited by a derived type in a descendant 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 itself "withs" a package A. (see figure 2.)

Consider first the packages on which the partition depends. They consist of the package B which "withs" package A:

```
package A is
   procedure PA ;
end A ;

with A ;
package B is
   procedure PB ;
end B ;
```

In Ada83 we would construct the ADT corresponding to A and then make the ADT for B exporting type for the state record for B containing the state variables of B together with an instance of the type exported by the transformed A.

In Ada9x, the translation can be done (of course) in exactly the same way, since the Ada83 programs will be valid in Ada9x. However, the aggregation of the states can receive language support by the use of record extension. First we make ADTs corresponding to the separate packages A and B and then form the aggregated state by type derivation. To make such extension legal, the type to be extended must be declared to be a "tagged" type.

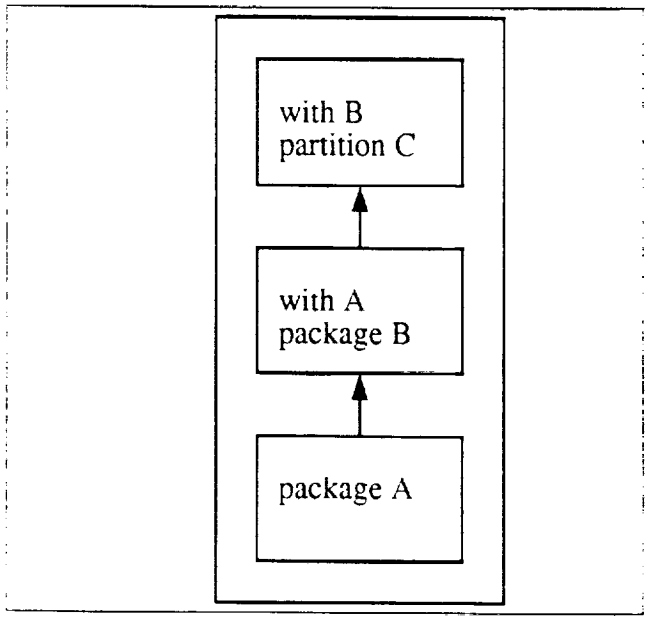


Figure 2: Schematic of the partition with its dependency graph.

```
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 ...
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 B_ADT;
```

Note that it is the type exported by package B which is inherited and extended; package A

provides a state type which is compounded into 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 these can be called with actual parameters of access type, provided they are given the the new mode **access** in the formal part.

```
package C_ADT is
   type C_State is tagged private;
   procedure PC (VC : in C_State);
private
   type C_State is ...
end C_ADT;

with B_ADT,
   C_ADT;
use B_ADT,
   C_ADT;
package Full_C_ADT is
   type Full_C_State is new C_State with private;
   type Full_C_Ptr is access Full_C_State;
   procedure PC (VC : in Full_C_State);
   procedure Create (VC : in out Full_C_Ptr);
   procedure Destroy (VC : in out Full_C_Ptr);
private
   type Full_C_State is new C_State with
      record
         VB : Full_B_State;
      end record;
end Full_C_ADT;
```

5 The Player Partition in Ada9x

We can now turn to the translation of the elements of the game, particularly the partition *Player* with its "withed" package *Boards*.

The first is the partition unit itself. The interface is converted to an ADT as follows:

```
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 thinks and plays until the game is over.

   type Player_state is private;
      -- Create simulates the operator new for the partition
      -- creation and supports the initialisation parameters.
      -- must abort the task instance as well as free storage.
   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
   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_Initial (P  : in Player);
   end Life_Type;

   type Player_State is
      record
           -- State variables derived from the body of the Player partition.
         The_Referee  : Referee_ADT.Referee;
         My_Turn  : T_Turn;
         T  : Life_Type;
      end record;
end Player_ADT;
```

The task body and the bodies of the operations *Create* and *Destroy* cannot be given yet, as they depend on the package *Boards*, whose operations they use. Thus next we provide the translation into an ADT of the package *Boards*. This ADT provides state and operations which will be used by the *Player* in executing its "life" task, so the package *Boards_ADT* must be compiled before the definition of the player body.

```
with Shared_Types;
use Shared_Types;
package Boards_ADT is

      -- This package provides the player with an instance of the board type:
      -- In this he maintains a record of all he has learned of his
      -- opponents' ship arrangement.  This package also provides the
      -- operation to construct the player's own battle arrangement
      -- to send to the referee.  Also the operations to actually conduct
      -- the game by selecting a move and recording the result.

   type Boards is private;
      -- Definition by the user (the player) of his/her ships'
      -- arrangement in the player's board.
   procedure User_Definition (B  : in out Boards; The_Board  : out T_Board);
      -- Operations based on the player's view of his/her opponent board.
   procedure Init_Opponent (B  : in out Boards);
```

```
   procedure Update (B : in out Boards; Tried_Move : in T_Move; Result : in T_Result);
   procedure Choose_Move (B : in out Boards; The_Move : out T_Move);
private
   type Boards is
      record
         Opponent_Board : -- as defined in the body of the AdaPT package Boards.
      end record;
      -- This is the state record for board instances.
end Boards_ADT;
```

It should be noted that the *Boards* type exported by *Boards_ADT* is a normal type and not an access type. The latter is only necessary to provide the effect of a partition, with its dynamic creation and capability of run-time switching. The access type references the aggregated state records as is shown in the package *Full_Player_ADT* below.

The extended player partition which represents the full partition with its associated package *Boards* is now given: it inherits its parent *Player_ADT* and enriches its state with the state of the *Boards_ADT*.

```
with Referee_ADT, Player_ADT, Boards_ADT,
   Shared_Types;
use Referee_ADT, Player_ADT, Boards_ADT
   Shared_Types;
package Full_Player_ADT is

      -- Player simulates the behaviour of a player.  After having
      -- its environment variables set up (through the initialisation
      -- parameters), it thinks and plays until the game is over.

   type Full_Player_State is new Player_state with private;
   type Player is access Full_Player_State with 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 Full_Player_State is new Player_State with
      record
           -- Aggregates State variables of Player with those of Board.
         B : Boards_ADT.Boards;
      end record;
end Player_ADT;
```

Finally, the bodies of the packages *Player_ADT* and *Full_Player_ADT* can be given. The task which defines the life makes reference to the *Boards* data structures which now form part of the *Full_Player_ADT*, which they can access in the state record. To make that possible it is necessary to pass a reference to that state record to the task at the *Create* time. Provision for this was made by providing a suitable entry in the task type *Life_type*.

```
with Boards_ADT ;
package body Full_Player_ADT is
        task body Life_type is
           Me : Player ;
        begin
           accept Set-Initial(P: Player) do
              Me := P ;
           end
        ...etc.-- illustrates how the task can access the state record
        end Life_type ;
end Full_Player_ADT ;
```

We should add that we do not necessarily consider that this structure is 'good' Ada9x. A programmer forming an Ada9x program component with these properties directly would arrive at exactly this structure.

5.1 Polymorphic types and conformant partitions

In section 4.2 we drew attention to the difficulty of constructing conformant partitions in Ada83 in view of the strong typing rules of the language. Ada9x supports a controlled degree of polymorphism, which can be utilised 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. However, to provide the polymorphic type, it is necessary first to define an empty (fully abstract) tagged type, and derive 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[4].

[4]Note: we have remarked in reports on the Ada83 translation that the AdaPT conformant partition has a problem that the overheads of providing for possible conformance is carried by all types, because there is no syntactic recognition of types which may have conformant peers. Limiting the polymorphism to tagged types avoids this difficulty in Ada9x.

```
package Abstract_B is
   type Empty_State is tagged
      record
         null;
      end record ;
   -- The type exported is to be polymorphic to reference items
   -- of the type and any derived type.
   type Class_Ptr is access Empty_State'CLASS ;
   procedure PB (VB : access Empty_State) is <>;
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_ADT is
   type B_State is new Abstract_B.Empty_State with private;
   type B_Ptr is access 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_ADT ;
```

```
with Abstract_B ;
package CB_ADT is
   type CB_State is new Abstract_B.Empty_State with private;
   type CB_Ptr is access CB_State ;
   procedure PB (VB : access CB_State) ;
   function Create return CB_Ptr ;
   procedure Destroy (VB : in out CB_Ptr) ;
private
   type CB_State is new Abstract_B.Empty_State with
      record
         -- State required by the implementation of CB.
      end record ;
end CB_ADT ;
```

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_ADT.Create ;
   Abstract_B.PB (VB) ; -- A call to B_ADT.PB.
   VB := CB_ADT.Create ;
   Abstract_B.PB (VB) ; -- A call to CB_ADT.PB.
end ;
```

Though rather verbose, we find this solution quite pleasing[5].

We now present the case of the conformant players introduced earlier in the same style:

```
package Abstract_Player is

      -- This is an abstract definition of Player's interface that
      -- defines a universal type representing the partition and
      -- the abstract operations which cannot be called and do not
      -- require a bodies.
   type Univ_Player is tagged
      record
         null;
      end record ;
   type Class_Player_Ptr is access Player'CLASS ;
end Abstract_Player ;



with Referee_ADT ,
   Shared_Types ,
   Abstract_Player ;
use Shared_Types ;
package Player_Impl is

      -- Full definition of Player's interface.
   type Player is new Abstract_Player.Univ_Player with private;
   type Player_Ptr is access Player ;
   function Create
          (A_Referee : Referee_ADT.Referee ;
           A_Turn : T_Turn) return Player_Ptr ;
   procedure Destroy (P : in out Player_Ptr) ;
private
   type Player is new Abstract_Player.Univ_Player with
      record
         -- Declaration of state corresponding to this particular
         -- implementation of a partition Player.
      end record ;
end Player_Impl ;



with Referee_ADT ,
   Shared_Types ,
   Abstract_Player ;
use Shared_Types ;
package Auto_Player_Impl is

      -- Full definition of Player's interface.
   type Auto_Player is new Abstract_Player.Univ_Player with private;
   type Auto_Player_Ptr is access Auto_Player ;
   function Create ,
          (A_Referee : Referee_ADT.Referee ;
           A_Turn : T_Turn) return Auto_Player_Ptr ;
   procedure Destroy (P : in out Auto_Player_Ptr) ;
```

[5]We would like to acknowledge the help of Offer Pazy of mapping team, in checking that this solution is indeed along the right lines

```
private
   type Auto_Player is new Abstract_Player.Univ_Player with
      record
         -- Declaration of state corresponding to this particular
         -- implementation of a partition Player.
      end record ;
end Auto_Player_Impl ;
```

And in the creating node, the following code could be found:

```
   A_Player : Abstract_Player.Class_Player_Ptr ;
      .
      .
begin
   A_Player := Player_Impl.Create (A_referee , A_Turn) ;
      -- the current player refers to a human player.
   A_Player := Auto_Player_Impl.Create( (A_referee , A_Turn) ;
      -- the current player refers to an automatic player.
end ;
```

5.2 The node in Ada9x

Having constructed types whose instances constitute the virtual nodes of a distributable program in Ada9x[6], they can now be assembled into one or more "supertypes" whose instances form the nodes of the program. Here is the outline of the type which encompasses all three units, as did the example of the AdaPT node above.

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 by the create operation of the node, calling the respective create operations of the partitions. Destroy works in a similar way.

```
package Node1_ADT is
   type Node1 is private;
   procedure Create (M : in out Node1);
   procedure Destroy (M : in out Node1);
private
   type State ;
   type Node1 is access State ;
end Node1_ADT ;


with Referee_ADT ,
   Player_ADT ,
```

[6] We should emphasise again that this is a program to implement the AdaPT concepts in Ada9x. It is not necessarily the way in which the same problem would be solved in the current mapping.

```
    AdaPT_Types ,
    Shared_Types ,
    UNCHECKED_DEALLOCATION ;
use Shared_Types ;
package body Node1_ADT is

    type State is
       record
          The_Referee  : Referee_ADT.Referee ;
          Player1 ,
          Player2  : Player_ADT.Player ;
       end record ;

    procedure Create  (M  : in out Node1) is
    begin
       M  := new State ;
       Referee_ADT.Create  (M.The_Referee) ;
       Player_ADT.Create  (M.Player1 , M.The_Referee , Turn1) ;
       Player_ADT.Create  (M.Player2 , M.The_Referee , Turn2) ;
    exception
       when others =>raise AdaPT_Types.NODE_ERROR ;
    end Create ;

    procedure Destroy  (M  : in out Node1) is
       procedure Free  is new UNCHECKED_DEALLOCATION
            (State , Node1) ;
    begin
       Referee_ADT.Destroy  (M.The_Referee) ;
       Player_ADT.Destroy  (M.Player1) ;
       Player_ADT.Destroy  (M.Player2) ;
    end Destroy ;

end Node1_ADT ;
```

As we saw before, the difference between a partition and a node lies in the presence or absence of a main procedure. The following is the main procedure required to make this node executable.

```
with Node1_ADT ;
procedure Main  is
    An_Instance  : Node1_ADT.Node1 ;
begin
    Node1_ADT.Create  (An_Instance) ;
end Main ;
```

After creating the instance of the state record, the procedure becomes completed, but does not return until all active partitions generated by the creation are terminated.

6 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 machine. To conform with Ada's requirements, it must

have the form of a procedure. When there are several nodes, to become binaries for allocation on different machines in a network, the transformation into Ada introduces the following new problems:

- identifier.
- node creation.
- remote communication.

6.1 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;
   .
begin
   .
   R.Copy_Board (A_Player, A_Board);
   .
end;
```

Until now, in the transformation of partitions and nodes into Ada 9X (as well as into Ada83), this reference has been implemented using an Ada **access** type.

```
with Shared_Types;
use Shared_Types;
package Referee_ADT is
   Referee is private;
   .
   procedure Copy_Board (Owner : in T_Turn; Board : in T_Board);
   .
private
   type Referee_State;
   type Referee is access Referee_State;
end Referee_ADT;
```

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 insufficient. 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 Ada 9X, the ADT's type must be extended adding a node identifier (i.e. program and machine identifiers).

```
with Shared_Types ;
   AdaPT_Types ;
use Shared_Types ;
package Referee_ADT is
   Referee_Ptr is private;
   type Referee is
      record
         Node_ID   : AdaPT_Types.T_Node_ID ;
         Reference : Referee_Ptr ;
      end record ;
      .
   procedure Copy_Board (Owner : in T_Turn ; Board : in T_Board) ;
      .
   procedure Create (R : in out Referee) ;
   procedure Destroy (R : in out Referee) ;
      .
private
   type Referee_State ;
   type Referee_Ptr is access Referee_State ;
end Referee_ADT ;
```

For the declaration of *Node_ID*, a new type *T_Node_ID* is specified in *AdaPT_Types*.

```
package AdaPT_Types is

   type T_Node_ID is
      record
         Process_ID ,
         Host_ID  : NATURAL ;
      end record ;

      -- AdaPT's predefined exceptions.

   Site_Error ,          -- Node name does not exist.
   Node_Inaccessible ,   -- Node is not available.
   Node_Error ,          -- Node elaboration cannot be completed.
   Partition_Error ,     -- Partition elaboration cannot be completed.
   Remote_Call_Error ,   -- Error during timed REC.
               -- Other communication errors.
   Communication_Failure  : exception;
end AdaPT_Types ;
```

The component *Node_ID* of *Referee* can be initialised at creation time (e.g. by calling the UNIX services to get a process identifier and a host identifier). Once a *Referee*'s instance has been created, *Node_ID* and *Reference* provide a "system-wide identifier" for the instance.

System-wide identifiers are necessary for nodes instances as well as for partition instances because both can be remotely referenced.

We note that after this modification the type *Referee* is no longer a **private** type. This allows the comparison of the *Node_ID* of different nodes or partitions to determine whether a call is remote or local.

```
if Self.Node_ID = R.Node_ID then
   -- Local call to R.
   R.Copy_Board (Self. My_Board) ;
else
   -- Remote call to R.
end if ;
```

In principle, the non-privacy of *Referee* in *Referee_ADT*'s specification should not lead to any misuse because all this code should be produced by an automatic transformation tool.

In the case of conformant partitions, the transformation into Ada83 defines a common type (e.g. *Universal_Referee.Partition*) for the instantiation of any conformant partition. This type specifies an identifier of the kind of partition inside the conformance set (i.e. *Selector*) and a reference to the partition instance itself (i.e. *Reference*). In a distributed environment, this reference must be converted into a system-wide identifier by adding a new component *Node_ID*.

```
package Universal_Referee is

   type Partition ;

   type T_Reference is access Partition ;

   type Partition is
      record
         Selector : NATURAL ;
         Node_ID : AdaPT_Types.T_Node_ID ;
         Reference : T_Reference ;
            -- The type used for the declaration of Reference
            -- could be any dummy access type.
      end record ;

   Undefined_Selector : exception ;

end Universal_Buffer ;
```

6.2 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 Ada 9X, we need to declare a **procedure** as a main program for the distinguished node.

```
with Node1_ADT ;
procedure Main_Node1 is
   N1 : Node1_ADT.Node1 ;
begin
   Node1_ADT.Create (N1) ;
end Main_Node1 ;
```

The non-distinguished nodes are nodes remotely created by other system's nodes. In the creation statement the location for the new node can be specified (by default it is the current machine). A nonexistent location must raise predefined AdaPT exception *SITE_ERROR*. The creator node has to wait until the creation is finished (i.e when a reference to the created node is returned). In our example, *Node1* creates *Node2*.

```
N2 : Node2 := new Node2'NODE (A_Network_Location) (B) ;
```

In a sense the creation of a node instance can be understood as a function call which returns a reference to the node instance created. Therefore, we could use an Ada **function** to implement the main program for *Node2*. A simplified version of such function is shown below.

```
with Node2_ADT ,
     Buffer_ADT ,
     AdaPT_Types ;
function Main_Node2 (R : Referee_ADT.Referee) return Node2_ADT.Node2 is
     N2 : Node2_ADT.Node2 ;
begin
     Node2_ADT.Create (N2 , R) ;
     return N2 ;
end Main_Node2 ;
```

In principle, this solution should work. As *N2* contains a task instance, even if the main function finishes and returns the reference, the program would run until that task terminates. 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.

```
with Node2_ADT ,
     Referee_ADT ,
     AdaPT_Types ,
       .
use AdaPT_Types ;
procedure Main_Node2 is

     procedure Get_Arguments (Creator_ID : out Node_ID ; R : out Referee_ADT.Referee) is
       -- It gets from the command line Creator_ID and R.

     procedure Return_Reference (To : in Node_ID ; Ref : in Node2_ADT.Node2) is
       -- Sends the reference to the node specified.

     procedure Return_Node_Error (To : in Node_ID) ;
       -- Sends the exception NODE_ERROR to the node specified.

     Creator_ID : Node_ID ;
     The_Referee : Referee_ADT.Referee ;
     N2 : Node2_ADT.Node2 ;
```

```
begin
   Get_Arguments (Creator_ID , The_Referee) ;
   Node2_ADT.Create (N2 , The_Referee) ;
   Return_Reference (Creator_ID , N2) ;
exception
   when NODE_ERROR =>Return_Node_Error (Creator_ID) ;
end Main_Node2 ;
```

A similar **procedure** must be defined for every non-distinguished node in the system.

As for the creator node (e.g. *Node1*), the creation of a non-distinguished node in Ada 9X 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).

A new package *AdaPT_System* can be defined to provide these services.

```
with AdaPT_Types ,
   Node2_ADT ,
   Referee_ADT ,
   .
use AdaPT_Types ;
package AdaPT_System is

      -- T_Location is an enumerate type that lists the names
      -- of the network nodes availables in the system.
   type T_Location is ..

   function Get_Node_ID return T_Node_ID ;
   function Create_Node2
            (Creator_ID  : in T_Node_ID ;
               Location  : in T_Location ;
               R  : in Referee_ADT.Referee) return Node2_ADT.Node2 ;
      -- It creates an instance of Node2 on the location
      -- specified returning a reference to the instance.  It
      -- may raise the following exceptions:
      -- SITE_ERROR when the location doesn't exist,
      -- NODE_ERROR when an elaboration error occurs during
      -- the node creation,
      -- COMMUNICATION_ERROR when there is a network
      -- communication error.
```

```
      .
      .
end AdaPT_System ;
```

AdaPT_System could provide a creation function for every non-distinguished node type defined in the system. We note that *T_Location* is declared here instead of in *AdaPT_Types* because it requires information about the specific AdaPT's system under transformation.

In its *Create* procedure, *Node1* will call *AdaPT_System.Create_Node2* for the remote creation of a *Node2*'s instance.

```
with UNCHECKED_DEALLOCATION ,
   AdaPT_System ;
package body Node1_ADT is
   procedure Create (N1 : in out Node1) is
   begin
      N1.Node_ID := AdaPT_System.Get_Node_ID ;
      N1.Reference := new Node1_State ;
        -- Local creation of a Referee's instance.
      Referee_ADT.Create (N1.Reference.B) ;
        -- Local creation of a Player's instance passing a
        -- Referee's reference.
      Full_Player_ADT.Create (N1.Reference.P , N1.Reference.R) ;
        -- Remote creation of a Node2's instance on
        -- A_Network_Location.
      N1.Reference.N2 := AdaPT_System.Create_Node2
            (N1.Node_ID , A_Network_Location , N1.Reference.R) ;
   exception
      when others =>raise AdaPT_Types.NODE_ERROR ;
   end Create ;
      .
      .
end Node1_ADT ;
```

6.3 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 Ada 9X, we have simply reused the main ideas of [DIADEM] which defines a "source-level" approach [7] that suits very well the transformation strategy followed until now. 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 layer[8]. For

[7] The code for supporting the remote communication is introduced by a transformation tool.

[8] Roughly speaking, the rendezvous layer corresponds to the ISO's session and presentation layers.

a more complete discussion of remote communication in translated AdaPT programs, please refer to the extended R3 report.

6.4 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. Details of the way in which such node images are allocated to machines can be found in the papers on AdaPT, while details of the communications mechanisms were discussed in detail in report R3.

7 Conclusions

The solution proposed here, based on a natural translation of the AdaPT concepts into Ada9x, does not match the approach proposed by the mapping team for distribution. Their solution has packages, called Remote Communication Interface (RCI) Packages which roughly serve in the role of our partitions. These are static units, and non-replicatable. Our use of data types as partition types is more dynamic, but may be less secure, being heavily dependent on the use of access variables.

We believe that our scheme of transforming AdaPT to Ada9x using ADTs shows that the use of data types as partitions, permitting inheritance to play a major role in designing distributed applications, makes a natural synthesis of these important parts of the language . At the present time, we feel that the elegance of making a coherent integration of the language in respect of the Object Oriented aspects, with derived types, and the distribution aspects is very attractive, and think it deserves careful consideration before the language is frozen with the version presently proposed. We therefore suggest that further careful study is justified to decide how the distribution aspects can best be married with the data typing.

In a later report we expect to make a critical comparison of the two styles, and make our recommendations.