Transforming AdaPT to ADA

REPORT 3

JOHNSON
GRANT
IN-61-CR
115103
P.29

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

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

April 27, 1991

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

NASA Johnson Space Center
Engineering Directorate
Flight Data Systems Division

(NASA-CR-190635) TRANSFORMING AdaPT TO Ada Interim Status Report No. 3 (Research Inst. for Computing and Information Systems) 29 p

N92-34196

Unclas

G3/61 0115103

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

INTERIM DRAFT REPORT

The RICIS Concept

The University of Houston-Clear Lake established the Research Institute for Computing and Information Systems (RICIS) in 1986 to encourage the NASA Johnson Space Center (JSC) and local industry to actively support research in the computing and information sciences. As part of this endeavor, UHCL proposed a partnership with JSC to jointly define and manage an integrated program of research in advanced data processing technology needed for JSC's main missions, including administrative, engineering and science responsibilities. JSC agreed and entered into a continuing cooperative agreement with UHCL beginning in May 1986, to jointly plan and execute such research through RICIS. Additionally, under Cooperative Agreement NCC 9-16, computing and educational facilities are shared by the two institutions to conduct the research.

The UHCL/RICIS mission is to conduct, coordinate, and disseminate research and professional level education in computing and information systems to serve the needs of the government, industry, community and academia. RICIS combines resources of UHCL and its gateway affiliates to research and develop materials, prototypes and publications on topics of mutual interest to its sponsors and researchers. Within UHCL, the mission is being implemented through interdisciplinary involvement of faculty and students from each of the four schools: Business and Public Administration, Education, Human Sciences and Humanities, and Natural and Applied Sciences. RICIS also collaborates with industry in a companion program. This program is focused on serving the research and advanced development needs of industry.

Moreover, UHCL established relationships with other universities and research organizations, having common research interests, to provide additional sources of expertise to conduct needed research. For example, UHCL has entered into a special partnership with Texas A&M University to help oversee RICIS research and education programs, while other research organizations are involved via the "gateway" concept.

A major role of RICIS then is to find the best match of sponsors, researchers and research objectives to advance knowledge in the computing and information sciences. RICIS, working jointly with its sponsors, advises on research needs, recommends principals for conducting the research, provides technical and administrative support to coordinate the research and integrates technical results into the goals of UHCL, NASA/JSC and industry.

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 Ada

Status Report 3

DRAFT

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

April 27, 1991

1 Introduction

This paper describes how the main features of the proposed Ada language extensions intended to support distribution, and offered as possible solutions for Ada9X can be implemented by transformation into standard Ada83. We start by summarising the features proposed in the paper [Gargaro et al,1990] which constitutes the definition of the extensions. For convenience we have called the language in its modified form **AdaPT** which might be interpreted as "Ada with partitions".

These features were carefully chosen to provide support for the construction of executable modules for execution in nodes of a network of loosely coupled computers, but flexibly configurable for different network architectures and for recovery following failure, or adapting to mode changes. The intention in their design was to provide extensions which would not impact adversely on the normal use of Ada, and would fit well in style and "feel" with the existing standard.

We begin by summarising the features introduced in AdaPT. These are defined in detail in the report cited [Gargaro et al, 1990].

1. **Partitions** A partition may be considered to constitute a "class" in the sense used in object oriented systems and languages. However, it is closely modeled on the

Ada package, presenting, in an interface specification, the items which are made available for its interaction with other system components. Thus its interface may contain procedures and functions, task declarations, and constants and exception declarations. It may not contain any object or type declarations. To help in defining the initial configuration a partition may have parameters (**in** parameters only), which are supplied by the program invoking the allocator when a new instance of the partition is created.

A partition is a library unit, and constitutes a type declaration. Other units may have **with** clauses to give them access to the definition in the library, and within the scope of the with clause they may declare variables of the type. However, the type is an implicit access type, and no instance of the partition is created by such a declaration. That is achieved by an assignment statement by copy from another variable of the same (access) type. Creation of new instances of a partition are obtained by **new** allocator statements, but these are not permitted in a partition. Such new partition instances are created during the configuration phase, in the definition of *nodes* which are described in the following paragraphs.

The use of library units "withed" by a partition lead to a special problem. Such packages may have "state", and consequently cannot be shared safely between different instances of a partition and between different partitions which may "with" the same unit. Thus, the semantics of with clauses for partitions is different from that for packages in a normal Ada program. The closure of the directed graph formed by the with clauses of a partition form part of the partition, and so are replicated as a whole with each replication of the partition. Each instance of a package or other object included in such a dependency graph, belongs therefore to one and only one partition. In contrast therefore to the public units described in 4 below, we sometimes refer to them as *non-public* units.[1]

2. **Nodes** Nodes differ very little from partitions. They too have features corresponding to those of packages; like partitions they have separate interfaces and bodies, and instance variables to reference them. However, nodes *can* create new instances of partitions and other nodes. Their role is to serve as units which will eventually be compiled and linked to form executable binary objects.

   The issue of system construction and start up and elaboration is described in AdaPT as a normal Ada main program call for a first selected node, called the *distinguished* node; this then "creates" others and so recursively until the whole system is elaborated.

3. **Conformant partitions** To support the provision of changed modes in a program,

[1]The word *private* has, of course, other connotations in Ada, including AdaPT.

particularly as a technique for recovery following failure of part of the system, partitions can have "peers" which have *identical* interfaces but different bodies. In object oriented terminology they would be of the same "type", possibly one a subtype of the other, but capable of providing the same set of actions for a client, albeit with different effect. As we shall see, this feature is difficult to reconcile with Ada's strong typing. In exact analogy with the idea of conformant partitions, it is proposed to support conformant nodes.

4. **Public Units** Partitions are permitted to share information, especially type information to give the types of the messages which form the parameters of sub-programs and task entries in the partition interfaces. Such sharing is permitted by sharing constant state packages; in order to enable the compiler to check that the "constant state" requirement is correctly followed, such shared units are called *publics.* Types in public units may be private, and may be defined along with operations on them so that they are "abstract data types".

This summary identifies the main features for which Ada translations of AdaPT must be defined.

2 DRAGOON and AdaPT

The structures proposed for Ada in the AdaPT extensions were selected in part as a result of experience with the language DRAGOON [See, for example, JOOP March 1991]. This language offers a fully object-oriented language, based on Ada but extending it through the use of a class concept and supporting inheritance and polymorphism. Objects which are class instances may be passive, changing their state only upon calls from other objects, or may be active, containing an internal thread of control, which executes concurrently with others. Objects are considered both as units of system construction and modularity, and indeed of re-use, and also as units of allocation in the network when the system is configured to execute in a network. An important aspect of the distributed case involves the idea that an object can inherit the run-time support for some computer system, so becoming an "executable class". Instances of such a class are executable binary objects.

In proposing the AdaPT extensions to Ada, it was felt that class inheritance provided in DRAGOON went further than was likely to appeal to the Ada community, since undoubtedly the philosophy of the 9X development would be one of keeping the impact on existing Ada users to a minimum to overcome perceived shortcomings. The *partition* was proposed to capture the essential nature of the class as a unit of modularity and allocation without introducing the largely orthogonal aspect of inheritance. However, the *node* in AdaPT does to some extent serve in the role of the executable class in DRAGOON; node instances are run-time binaries compiled and linked to run on the appropriate machines. It is worth noting that in both DRAGOON and AdaPT the use of non- homogeneous networks is a small step (no dependence on shared memory exists), though in neither case has it so far been proposed to press forward in that direction.

DRAGOON is supported by software which transforms the programs into conventional Ada. At one time it was thought possible that the DRAGOON preprocessor might be modified for use in translating AdaPT codes, but eventually it seemed that, despite the relationship which exists between the languages, they are sufficiently different that it was not a useful direction to proceed. Rather in this paper we describe a design for an AdaPT translation tool, and how the translations can be done by hand, using the experience from the DRAGOON work.

3 Abstract State Machines vs Abstract Data Types

To understand the way in which partitions are handled in our Ada translation, it is necessary first to be familiar with the idea of an *Abstract Data Type* and with the related notion of an *Abstract State Machine.* This section reviews these concepts and compares them.

Abstract State Machines

An ASM is a software component which possesses some state (which may be hidden or visible). Operations that access and modify that state are provided in its interface. A simple example of an ASM in Ada is the following package:

```
package Variable_ASM is
    procedure Write (Value : in NATURAL);
    procedure Read (Value : out NATURAL);
end Variable_ASM;
```

The state, which here has its simplest possible form, a single variable, is encapsulated in the package body and is accessed by users of the package through the interface procedures, read and write.

```
package body Variable_ASM is
```

```
    The_variable : NATURAL;
    procedure Write (Value : in NATURAL) is
    begin
        The_variable := Value;
    end Write;
    procedure Read (Value : out NATURAL) is
    begin
        Value := The_variable;
    end Read;
end Variable_ASM;
```

Abstract Data Type

An ADT is more general[2] than an ASM in that it defines a *type* and a set of operations which can be applied to objects of that type. Conceptually, the type represents a state. An Ada **private record** type is generally preferred for the corresponding type declaration. A **record** type is used for generality, although here the record contains only a single variable, in most cases there will be a number of variables forming the state.

```
package Variable_ADT is
    type Variable is private;
    procedure Write (V : in out Variable; Value : in NATURAL);
    procedure Read (V : in out Variable; Value : out NATURAL);
private
    type Variable is
        record
            The_variable : NATURAL;
        end record;
end Variable_ADT;
```

This package is associated with a body which defines the operations of the procedures:

```
package body Variable_ADT is
    procedure Write (V : in out Variable; Value : in NATURAL) is
    begin
        V.The_variable := Value;
    end Write;
```

[2]An ADT may be used to construct an ASM. An ASM cannot be used to simulate an ADT.

```
    procedure Read (V : in out Variable; Value : out NATURAL) is

    begin
        Value := V.The_variable;
    end Read;
end Variable_ASM;
```

Users of this package can declare objects (instances) of type *Variable*. Every such object, declared to be of the ADT type, is a different instance of the state.

State instances can be manipulated through the operations provided by the ADT. The following is a fragment of a possible *Variable_ADT* user.

```
with Variable_ADT;
procedure Example is
      V1 : Variable_ADT.Variable;
      V2 : Variable_ADT.Variable;
begin
      .
      Variable_ADT.Write (V1, 8);
      Variable_ADT.Write (V2, 8);
      .
end Example;
```

Relating an ASM to a Corresponding ADT

Any ASM can be related[3] to an ADT. The basic converting strategies are the following:

- any element that defines part of the state in the ASM must become a component of the ADT's (record) type,
- all the operations provided in the ASM's interface gain an extra parameter which is an object of the ADT's type,
- any ASM's caller must declare an ADT's object and pass it in its calls to the ADT.

[3]We do not use the term *translation* because the resulting ADT as such does not respect the ASM's semantics. Instead of maintaining the state at the callee level (as it happens in an ASM), the state is kept inside the caller. Every caller possess a different state copy.

To define **the ADT's type**, the elements with state which must be taken into account are the objects (variables) and the tasks declared both in the specification and in the body of the ASM. The objects are straightforwardly converted into components of the state record type. In our *Variable* example, the object *The_Variable* is transformed into a component of the *Variable_ADT*'s type. As for the tasks, they first become task types and then their instances are included as components of the state record type. To illustrate this case, we convert our *Variable_ASM* into a *Protected_Variable_ASM* by defining a task in its body.

```
package body Protected_Variable_ASM is
     The_variable : NATURAL;
     task The_monitor is
          entry Start_Write;
          entry End_Write;
          entry Start_Read;
          entry End_Read;
     end The_monitor;
     task body The_monitor is
          .
          .
     end The_monitor;
     procedure Write (Value : in NATURAL) is
     begin
          The_monitor.Start_Write;
          The_variable := Value;
          The_monitor.End_Write;
     end Write;
     procedure Read (Value : out NATURAL) is
     begin
          The_monitor.Start_Read;
          Value := The_variable ;
          The_monitor.End_Read;
     end Read;
end Protected_Variable_ASM;
```

The corresponding *Protected_Variable_ADT* will have the following specification:

```
package Protected_Variable_ADT is
     type Variable is private;
     procedure Write (V : in out Variable; Value : in NATURAL);
     procedure Read (V : in out Variable; Value : out NATURAL);
private
```

```
    task type The_monitor_Type is
        entry Start_Write;
        entry End_Write;
        entry Start_Read;
        entry End_Read;
    end The_monitor_Type;

    type Variable is
        record
            The_variable : NATURAL;
            The_monitor : The_monitor_Type;
        end record;
end Protected_Variable_ADT;
```

When the ASM contains elements with visible state (declared in its interface), the use of a **private** Ada type for the ADT type declaration implies an extra overhead. New procedures must then be provided for access to objects that were directly visible in the ASM and which are now hidden inside the ADT type. In the same way, extra procedures are necessary to give access to task entries when the corresponding task is hidden in the private part of the ADT. Consequently, any reference inside a caller to those visible state elements must be converted into a procedure call. The code of any users of the ASM's will also be affected. In such cases, the use of a **private** type may be considered inappropriate.

Once the ADT's type is constructed, we define **the ADT's interface operations.** The ADT's object is added as an extra parameter to the operations provided in the ASM's interface. In our *Variable* example, the procedures *Write* and *Read* declarations are transformed from:

```
package Variable_ASM is
    procedure Write (Value : in NATURAL);
    procedure Read (Value : out NATURAL);
end Variable_ASM;
```

to

```
package Stack_ADT is
    .
    procedure Write (V : in out Variable; Value : in NATURAL);
    procedure Read (V : in out Variable; Value : out NATURAL);
    .
```

```
end Variable_ADT;
```

The subprograms inside the ASM's body must also be adapted to deal with the extra parameter. For example, in the *Protected_Variable_ADT* the *Write*'s body has become:

```
procedure Write (V : in out Variable; Value : in NATURAL) is
begin
     V.The_monitor.Start_Write;
     V.The_variable := Value;
     V.The_monitor.End_Write;
end Write;
```

In this way, the operations manage a particular state instance passed to it by the caller, instead of the state maintained inside the ASM's.

The result of this two-step translation is an ADT. No more transformation on the ASM is needed. Only some adaptations are still required on **the ASM's callers**. Any unit that uses the ASM is transformed to deal with the ADT, by applying the three following modifications:

- the substitution of the ASM by the ADT in the **with** clause,
- the inclusion of a object declaration of the ADT type[4],
- the passing of the declared ADT object as a new parameter in every call to the ASM.

A possible *Variable_ASM*'s caller is:

```
with Variable_ASM;
procedure Example is
begin
        .
     Variable_ASM.Write (8);
        .
end Example;
```

[4]From the semantic point of view, the ADT object should be declared in a declarative part that gives it the same scope than the one established by the original **with** clause. Normally this means a declaration at the highest level.

After applying the three steps described above, we will obtain:

```
with Variable_ADT;
procedure Example is
    V : Variable_ADT.Variable;
begin
    .
    Variable_ADT.Write (V, 8);
    .
end Example;
```

4 Transforming a Partition into an ADT

The partition is a new construct introduced by AdaPT that captures the virtual node type concept (the minimum unit of distribution and of dynamic node creation). A partition declaration is equivalent to an **access** type declaration. Therefore a **new** operation is required to create a partition object. A partition instance is an ASM.

Like a package, a partition may have a specification (its interface) and a body. Initialization parameters (only **in**) can be defined for configuration purposes. Mainly three constraints exist on the specification declarations due to the distribution. First, to avoid remote memory accesses, no object declaration is allowed. Second, type declarations are also excluded because of the dynamic type checking that they would require. The Ada strong typing would imply that the different instances of a partition would define different type declarations. Public units are provided for those type declarations. Finally, no partition creation is allowed inside a partition (partitions can refer to other partitions but they cannot create them).

The term partition is "overloaded" and may also be used to name the larger entity consisting of the partition unit itself with all the elements in the "closure" of its dependency graph. The dependency graph consists of the units implied by **with** clauses appearing in the partition's context clauses but not including:

- public units[5],
- other partitions.

[5]Public units are normally shared between various partitions and then do not belong to any of them. Public units do not require any transformation due to their constant state character.

As explained above, conformant partitions can be defined for the purpose of supporting mode changes. A partition that is conformant to another one, has the same interface but a different body (and most often a different set of "withed" units).

4.1 The Partition Closure

In our transformation description, we assume that all units are correct[6]. The conversion strategy is based again on the ADT's concept. A simple partition that provides a buffer will be used as an example in the following sections.

```
partition Buffer is
    Max_Num_Items : constant NATURAL := 50;

    procedure Put (Item : in  INTEGER);
    procedure Get (Item : out INTEGER);
    function Num_of_Items return NATURAL;
end Buffer;


partition body Buffer is
        -- Linked list implementation of a buffer.
    type Cell;
    type Link is access Cell;
    type Cell is
        record
            Data : INTEGER;
            Next : Link;
        end record;
    First_Item,
    Last_Item : Link := null;
    Num_Items : NATURAL := 0;
        .
        .
 end Buffer;
```

Every non constant state unit[7] belonging to the partition closure is converted into an ADT, as described in previous sections. An instance of that state is thus added to the

[6]Before any transformation, the syntax of all units (publics, non publics, partitions and nodes) must be checked. Besides, the semantic constraint on the partition creation inside a partition has to be verified.

[7]Units without state or with constant state do not require any transformation.

state declarations of any unit which "withs" it. In this way, any state defined in any of those units is transferred to its caller until the partition itself is reached.

The partition is also translated into an ADT. As a partition defines an **access** type, the corresponding ADT's type must be an Ada **access** type[8]. To simulate the **new** operation necessary to create a partition object, a procedure *Create* must be defined in the ADT's interface. This procedure has an **in out** parameter which is the ADT's object itself, and **in** parameters matching with the partition initialization parameters.

```
package Buffer_ADT is
    Max_Num_Items : constant NATURAL := 50;
    type Buffer is private;
    procedure Put (B : in out Buffer; I : INTEGER);
    procedure Get (B : in out Buffer; I : out INTEGER);
    function Num_of_Items (B : in Buffer) return NATURAL;
    procedure Create (B : in out Buffer);
private
        -- Linked list implementation of a buffer.
    type Cell;
    type Link is access Cell;
    type Cell is
        record
            Data : INTEGER;
            Next : Link;
        end record;
    type Buffer_State is
        record
            First_Item,
            Last_Item : Link := null;
            Num_Items : NATURAL := 0;
        end record;
    type Buffer is access Buffer_State;
end Buffer_ADT;
```

The final result of all these transformations is a unique state representation type of the whole closure. Therefore every partition object declaration will define a different state.

[8]In a normal transformation of an ASM to an ADT, an access type is not required as shown in the *Variable* example. The non-public units in the dependency graph give rise only to variables of non access data type.

Generally the units belonging to the closure can be transformed in a one-to-one unit mapping way. However sometimes this can result in an inconsistent situation. An example is a partition *P* that uses two non constant state units *A1* and *A2* which both use a common non constant state unit *B*.

B;

with B; A1;   with B; A2;

with A1, A2: P;

Applying the already explained strategy to the above example, the translation steps are:

1. *B* will be converted to *B_ADT*,
2. *A1* to *A1_ADT* which will contain an instance of *B_ADT*'s state within its state representation type,

```
type A1 is
    record
        Ref_to_B : B_ADT.B;
        -- Components representing proper A1's state.
           .
    end record;
```

3. *A2* to *A2_ADT* which will also contain an instance of *B_ADT*'s state inside its state representation type,

```
type A2 is
    record
        Ref_to_B : B_ADT.B;
        -- Components representing proper A2's state.
           .
    end record;
```

4. *P* to *P_ADT* which will include both *A1_ADT*'s and *A2_ADT*'s state in its state representation type.

```
type P is
    record
        Ref_to_A1 : A1_ADT.A1;
        Ref_to_A2 : A2_ADT.A2;
        -- Components representing P's own state.
            .
    end record;
```

This transformation leads to an inconsistency, since following the state references from *P* to *B*, it appears that as *P* refers to *A1* and to *A2* separately and each has its own reference to *B*, therefore *B* is referenced twice instead of once. This problem can be solved in two alternative ways. First, **access** types could be used to represent the ADT's types for *A1_ADT*, *A2_ADT* and *B_ADT*. The *Create* operation of an *P_ADT*'s instance could then check to obtain a unique reference to *B_ADT* (same value inside *A1_ADT*'s and *A2_ADT*'s states). This solution can support the one-to-one unit mapping scheme. However it implies a computing overhead and it is a quite complicated solution. The second solution recognizes that this problem would not exist if the dependency graph was a tree. It arises only when the closure has a general acyclic graph structure. However, this graph can be converted into a tree by joining units. In such a solution, only one ADT (*A1A2_ADT*) would be built from *A1* and *A2*. This would include a unique reference to *B_ADT*.

```
type A1A2 is
    record
        Ref_to_B : B_ADT.B;
        -- Components representing A1's and A2's proper state.
            .
    end record;
```

The disadvantage is quite evident. The one-to-one unit mapping scheme is no longer fully maintained. Otherwise the solution is far simpler than the first one and consequently it is preferred.

4.2 The Conformant Partitions

Originally, conformant partitions were thought of as a tool for providing degraded modes. The intent was to declare a partition which defines a certain behaviour and then conformant partitions which might serve for recovery following a partial system failure. This was implicitly distinguishing two kind of partitions (the first declared partition and the conformant partitions to that one). Now, we have realized that conformant partition could also be used for non recovery purposes like algorithm tests. In such cases, all conformant partitions (included the first declared one) have the same purpose and are equals. In this way, the conformance relation is understood as a relation between partitions sharing a common interface but having separate bodies.

To explain the transformation of conformant partitions into Ada83, we introduce a partition *Conf_Buffer* conformant to the partition *Buffer* already cited.

```
partition Conf_Buffer is Buffer ;
```

By definition, *Conf_Buffer* has the same interface as *Buffer*. *Conf_Buffer*'s body could be an array implementation of a buffer.

```
partition body Conf_Buffer is
        -- Array implementation of a buffer.
    subtype Buffer_Index is NATURAL range 1 .. Max_Num_Items;
    Items_Buffer : array (Buffer_Index) of INTEGER;
    I,
    J : Buffer_Index := 1;
    Num_Items : NATURAL range 0 .. Max_Num_Items := 0;
        .
        .
end Conf_Buffer;


package Conf_Buffer_ADT is
    Max_Num_Items : constant NATURAL := 50;
    type Buffer is private;
    procedure Put (B : in out Buffer; I : NATURAL);
    procedure Get (B : in out Buffer; I : out NATURAL);
    function Num_of_Items (B : in Buffer) return NATURAL;
    procedure Create (B : in out Buffer);
private
    subtype Buffer_Index is NATURAL range 1 .. Max_Num_Items;
    subtype Items_Index is NATURAL range 0 .. Max_Num_Items;
    type Buffer_Imple is array (Buffer_Index) of NATURAL;
    type Buffer_State is
        record
            Items_Buffer : array (Buffer_Index) of INTEGER;
            I,
            J : Buffer_Index := 1;
            Num_Items : Items_Index := 0;
        end record;
    type Buffer is access Buffer_State;
end Conf_Buffer_ADT;
```

It may be noted that the actual definitions of types in the private part might be deferred to the package body; only an incomplete type definition for *Buffer_State* and the declaration of *Buffer* need appear in the partition.

```
package Conf_Buffer_ADT is
    .
private
    type Buffer_State;
    type Buffer is access Buffer_State;
end Conf_Buffer_ADT;
```

Here they are shown in full for convenience.

Conformant partitions introduce in Ada two new features difficult to reproduce in Ada83. Those features are quite similar to the object oriented concepts of *polymorphism* and *dynamic binding*. For their implementation, we have used the experience gained in DRAGOON's translation to Ada.

Polymorphism in AdaPT

Like subclass declarations of a common parent in an object oriented language, conformant partition declarations are intended to imply compatible type declarations. Thus, assignment statements may be executed between conformant partition variables[9]. In other words, a conformant partition variable may point to different conformant partition instances during its life.

```
        Current_Buffer : Buffer;
        A_conf_Buffer : Conf_Buffer;
```

[9]We note that a partition object, structurally speaking, is "a partition variable" that points to "a partition instance" (because a partition defines an access type which refers to an anonymous type). In this section, we will use both terms to facilitate the understanding of the polymorphism concept.

```
        .
begin
        -- The creation of the partition variables.
    Current_Buffer := new Buffer'PARTITION ;
    A_conf_Buffer := new Conf_Buffer'PARTITION ;

        .
        -- The use of Current_Buffer which refers to a partition Buffer.

        .
        -- A mode change.
    Current_Buffer := A_conf_Buffer;
        -- The use of Current_Buffer which refers to a partition Conf_Buffer.

        .
end ;
```

In **DRAGOON**'s translation into Ada, a common type declaration for all classes is used to implement this feature. This type called *APPLICATION.OBJECT* is a list whose first element has the same structure for every class instance. Next elements in the list provide the state information for every class or subclass appearing in the inheritance sequence. As the state information varies depending on the class represented, the elements in the list, except the header, may be of different type. To implement such a structure in Ada, the Ada strong typing must be overridden using UNCHECKED_CONVERSION[10]. Besides the state information, any element in the list includes a selector value, named *OFFSPRING_NO*, that determines what kind of element is its successor. Using both, the *OFFSPRING_NO* and UNCHECKED_CONVERSION, the list can be built and managed.

In **AdaPT**, the conformance relation establishes a limited kind of polymorphism between the conformant partitions. To simulate this relation in Ada83, some of the mechanisms employed by DRAGOON's translation can be reused. For instance, we introduce an extra type declaration, called *Partition*, for each conformant relation. Unlike the type *APPLICATION.OBJECT* which relates all classes existent in a DRAGOON's application, the type *Partition* will only associate the partitions belonging to the same conformance relation. Different conformance relations (e.g *Buffer*, *Consumer*) in an AdaPT's system will imply different types (e.g *Universal_Buffer.Partition*, *Universal_Consumer.Partition*) in the corrresponding Ada system. In this way, the incompatibility of different conformance relations is clearly stated and maintained. The

[10]DRAGON's experience demonstrates that the use of UNCHECKED_CONVERSION in a controlled way as a result of an automatic translation should not raise any problem. However, we recognize that it does not reflect a good Ada programming style.

type *Partition* is an Ada **record** with two components, named *Selector* and *Reference*. *Reference* is of type **access** and may point to any of the ADT's types produced by the conformant partitions transformation. To obtain this effect, the Ada Strong Typing is overridden by using UNCHECKED_CONVERSION. *Selector* is a discrete value which identifies what kind of partition[11] from the conformant set is currently referred.

```
package Universal_Buffer is
     type Partition;
     type T_Reference is access Partition;
     type Partition is
          record
               Selector : NATURAL;
               Reference : T_Reference;
          end record;
     Undefined_Selector : exception;
end Universal_Buffer;
```

Every ADT, representing in Ada83 one of the conformant partitions, exports a constant *Selector* value that identifies itself.

```
with Universal_Buffer;
package Buffer_ADT is
       .
     Selector : constant NATURAL := 0;
       .
     procedure Create (B : in out Universal_Buffer.Partition);
       .
end Buffer_ADT;
```

The ADT uses this value to initialize the *Selector* component of any partition variable that it creates.

```
package body Buffer_ADT is
```

[11]*Selector* is used to distinguished the different partitions involved in a conformance relation (e.g. *Buffer* and *Conf_Buffer*). Multiple variables of a particular partition (e.g *Buffer*) will imply the same *Selector* value and different *Reference* values.

```
      .
   function Buffer_to_Reference is new UNCHECKED_CONVERSION
           (Buffer, Universal_Buffer.Partition);

      .
   procedure Create (B : in out Universal_Buffer.Partition) is
        Buff : Buffer := new Buffer_State;
   begin
        B.Selector := Selector;
        B.Reference := Buffer_to_Reference (Buff);
   end Create;

      .
end Buffer_ADT;
```

Concluding, the AdaPT 's example of polymorphism is transformed into the following Ada83 code:

```
      Current_Buffer,
      A_conf_Buffer : Universal_Buffer.Partition;

      .
   begin
          -- The creation of the partition variables.
      Buffer_ADT.Create (Current_Buffer);
      Conf_Buffer_ADT.Create (A_Conf_Buffer);
          .
          -- The use of Current_Buffer which refers to a partition Buffer.

          .
          -- A mode change.
      Current_Buffer := A_conf_Buffer;
          -- The use of Current_Buffer which refers to a partition Conf_Buffer.

          .
   end ;
```

Dynamic Binding in AdaPT

As already explained, the conformance relation associates an interface with a set of implementations. A call to this interface has to be dynamically bound to the correct implementation. In other words, depending on the current conformant partition referenced by the partition variable, a different partition body must be called.

```
        Current_Buffer : Buffer;
        A_conf_Buffer : Conf_Buffer;

        .

    begin
            -- The creation of the partition variables.
        Current_Buffer := new Buffer'PARTITION ;
        A_conf_Buffer := new Conf_Buffer'PARTITION ;

            .

            -- The use of Current_Buffer which refers to a partition Buffer.
        Current_Buffer.Put (8);
            -- A mode change.
        Current_Buffer := A_conf_Buffer;
            -- The use of Current_Buffer which refers to a partition Conf_Buffer.
        Current_Buffer.Put (8);
    end ;
```

This feature of AdaPT is similar to the object oriented concept of *dynamic binding*. In DRAGOON, a call to a class method is dynamically bound to the class instance currently pointed to by the class variable.

Two alternative solutions have been considered to implement this behaviour in Ada83. Both involve the selection of the appropriate conformant partition body by consulting the *Selector* component of the *Partition* variable. These solutions differ in where the selection is made.

Indirect calls via an extra unit

In this solution an extra level of call indirection would be introduced between the caller and the possible callees (i.e. the conformant partitions bodies) to execute the selection. This extra level would define a specification twin to the unique conformant partition specification. Its body would direct the calls to the different partition bodies. In some ways, this solution reflects perfectly the concept of conformance, emphasizing the existence of a unique interface common to all conformant partitions. However, this solution has two important disadvantages. First, the inclusion of an extra level of call indirection implies a run-time overhead. Second, a task entry defined in the conformant partition interface can no longer be directly called. We remember that a task declared in a partition specification is converted into a task type and afterwards included in the corresponding ADT's type. In this way, no procedures are required to give access to the task entries and the transformation strategy of moving any state object into the ADT's type is respected. To illustrate this, we define *Sync_Buffer* which provides a synchronous buffer.

```
partition Sync_Buffer is
    Max_Num_Items : constant NATURAL := 50;

    task Buffer is
        entry Put (Item : in  INTEGER);
        entry Get (Item : out INTEGER);
        entry Num_of_Items (Num : out NATURAL);
    end Buffer;

end Sync_Buffer;

partition Conf_Sync_Buffer is Sync_Buffer;
```

The corresponding ADT will be:

```
with Universal_Sync_Buffer;
package Sync_Buffer_ADT is
    Max_Num_Items : constant NATURAL := 50;

    task type Buffer_Type is
        entry Put (Item : in  INTEGER);
        entry Get (Item : out INTEGER);
        entry Num_of_Items (Num : out NATURAL);
    end Buffer_Type;

    type Buffer_State is
        record
            T : Buffer_Type;
        end record;

    type Buffer is access Buffer_State;

    procedure Create (B : in out Universal_Sync_Buffer.Partition);

        .

end Sync_Buffer_ADT;
```

The extra level of call indirection would need to define in its specification procedures to give access to the task entries.

```
procedure Put (B : in out Buffer; Item : in  INTEGER);
procedure Get (B : in out Buffer; Item : out INTEGER);
procedure Num_of_Items (B : in out Buffer; Num : out NATURAL);
```

.This clearly does not match with the transformation strategy followed so far.

Selection by the caller

In the second solution, the selection of the appropriate partition body may be made by the caller itself. A call in AdaPT to a conformant partition subprogram like the following:

```
Current_Buffer.Put (8);
```

will be converted into the Ada83 code:

```
case Current_Buffer.Selector is
     when Buffer_ADT.Selector =>
          Buffer_ADT.Put (Current_Buffer, 8);
     when Conf_Buffer_ADT.Selector =>
          Conf_Buffer_ADT.Put (Current_Buffer, 8);
     when others =>
          raise Universal_Buffer.Undefined_Selector;
end case ;
```

We note that now the the parameter passed to the ADT's subprograms is of type *Partition*. To access the ADT's state, the ADT's subprograms must first obtain the partition instance.

```
procedure Put (B : in out Universal_Buffer.Partition; Item : in NATURAL) is
     An_instance : Buffer;
begin
     An_instance := Reference_to_Buffer (B.Reference);
          .
end Put;
```

For this purpose, a new function *Reference_to_Buffer* has been included in the ADT's body.

```
function Reference_to_Buffer is new UNCHECKED_CONVERSION
          (Universal_Buffer.T_Reference, Buffer);
```

A call to a partition task entry will not be so straightforwardly implemented in Ada83. To the execute the call, the task reference is necessary. As a partition visible task is transformed into a component of the corresponding ADT's type, only the partition instance currently referred by the *Partition* variable can provide the task reference. In our *Sync_Buffer*'s example, a call to the task entry *Put* would be converted into:

```
case Current_Buffer.Selector is
     when Sync_Buffer_ADT.Selector =>
          A_buffer_inst := Sync_Buffer_ADT.Reference_to_Bufffer (Current_Buffer);
          A_buffer_inst.Put (8);
     when Conf_Sync_Buffer_ADT.Selector =>
          A_conf_buffer_inst := Conf_Sync_Buffer_ADT.Reference_to_Bufffer (Current_Buffer);
          A_conf_buffer_inst.Put (8);
     when others =>
          raise Universal_Buffer.Undefined_Selector;
end case ;
```

The new function *Reference_to_Buffer* must be specified in the ADT's interface when there is a visible task. Otherwise, it could stay hidden in the ADT's body.

At first glance, this solution may seem to establish too much dependence between the caller and the set of possible callees. However, we note that anyway a conformant partition caller depends on the set of conformant partitions. When a new partition is added to the conformant set, the caller's code should be updated whether it wants to use it or not. A real disadvantage of this solution is the overhead implied by the task entry calls. The call to obtain the partition instance (e.g. *Reference_to_Buffer*) implies an extra call which could be a remote one. In fact, the worse case would be the remote one, when the caller and the callee reside in different nodes of the network. As remote Rendez-vous are already themselves very expensive operations, we do not think that the overhead produced by getting the partition instance is so important. Anyway, for real-time systems with strong time requirements, the use of remote Rendez-vous should be avoided. Otherwise this solution has two great advantages. First, it fits well in a distributed environment with possible failures. Whenever a mode change is executed modifying the referred conformant partition, there is no need to use the previous referred partition (which could have been running in a node that now is unavailable). Second, this solution shows clearly the management of the conformant partitions. Being non transparent, this solution seems more easy to understand and to use. We have chosen it for our transformation of the conformant partitions into Ada83.

Finally, we note that DRAGON's implementation in Ada of dynamic binding has not been mentioned as a possible alternative for the conformance relation support. In DRAGOON's translation, method calls are dynamically bound to their implementation by following the inheritance sequence until the correct class is reached. This procedure does not fit well with the conformance relation which is a one-level relation, clearly structurally different from the inheritance relation. Moreover, as the search for the method implementation goes through the bodies of all classes involved in the inheritance sequence, the procedure depends critically on the availability of those bodies. In AdaPT's transformation into Ada83, such a dependence between the conformant partition bodies cannot be established. It would run up against the purpose of the conformance relation in supporting mode changes[12].

Concluding remarks on conformance

The implementation of the conformance relation in Ada83 increases the code and the declarations required to simulate the partitions behaviour. With respect to the transformation, this fact does not have too much importance as probably the transformation would be an off-line tool. However, concerning the Ada83 system produced, the implementation of the conformance relation affects the system efficiency, mainly in the calls to partitions. We remember that to deal with the conformance relation, each partition call implies:

- identifying the kind of partition currently referenced,
- obtaining the corresponding partition instance.

The cost may even include extra calls over the network.

As seen in earlier sections of this report, implementation of a partition not involved in any conformance relation does not require this extra code. Unfortunately, however, it is not possible to know, when a given partition is transformed whether it will eventually become just the first of a set of conformant "peers". It is therefore necessary to treat every partition in the general way. Enabling calls to "non-conformant" partitions to be handled in a more efficient way could be achieved using some transformation tool optimisation or by defining a new reserved word in AdaPT to identify "non-conformant" partitions. Normal ADTs could then be produced, without including the modifications due to the conformance relation, thus avoiding the unnecessary run-time overhead. Unfortunately, in the case of a later redesign involving the introduction of a conformance relation for one

[12]Mode changes can occur as a result of a node failure and then imply the unavailability of one of the conformant partition bodies.

of those "non-conformant" partitions, the generation of the corresponding Ada83 system would require more effort because of the updating of all old references. At the current stage of the project, this topic remains open.

5 ADTs and Virtual Nodes

It is interesting to note that the translation of partitions into ADT packages leads to the recognition that such packages are already structures which provide in conventional Ada the behaviour needed for Virtual Nodes. There have been many previous attempts to find ways of constructing Ada programs for distribution. For example, DIADEM [], ASPECT[] etc... Such solutions have usually lacked the degree of flexibility required for reliable real time systems. In fact, all these efforts overlooked the possibility of using an ADT package as a virtual node. It is perfectly possible to write Ada code with the properties of the translated AdaPT directly, once the understanding of what is being done has been grasped. Partitions may be a luxury which can be dispensed with as a language feature, though they would remain a very useful concept to guide the development of correctly structured software.

It must be noted, however, that a package exporting a type can have many properties which would make it unsuitable for use as a virtual node class. Thus the addition of a language concept would constrain programmers to write well structured partitions, but we consider that an important outcome of this work has been to recognize a new way of programming distributed systems in conventional Ada. In a sense AdaPT should remain as an important methodology even if software to support it is never written.

6 Location of Generated Code

Since the purpose of the AdaPT scheme is to create programs suitable for execution on distributed hardware, it is important for system efficiency that the state information and the operations which update it are stored in the same machine[13]. This may be different from the machine holding the instance variable which references it. It is therefore worth noting that the space for the state record is provided by the transformed code in the create operation of the ADT. Within the body of that operation, an Ada **new** allocator will actually create the space on the heap of its current machine. This is the intended effect provided there is only one instance of the partition, or at least that all such instances are created in the same node.

[13]Indeed this is what virtual nodes are intended to ensure.

If several partition instances are required in different nodes (which will be a common situation) the ADT derived from its declaration must be replicated in each node and provision made to ensure that the create operation is executed in the appropriate node. This will be further discussed in section 7.2.

7 Structuring the System in Nodes

A full report on the structure of a network of nodes will be presented at a later date. Here we are limiting this account to an outline of the general principles.

A node has a structure identical to that of a partition. It must fulfil the same constraints except that a node is allowed to create partitions as well as nodes. Therefore, like partitions, nodes are transformed into ADT's. However, partitions and nodes have different purposes. Unlike partitions that reflect the unit of distribution, nodes support the concept of configuration. Inside an application, a node will be used to express a unit of network allocation. The purpose of a node is to become, eventually, 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. This procedure is called from the networking system and forms, as we shall see, part of the structure required to enable the correct creation of the nodes at system elaboration.

The transformation of nodes into Ada83 introduces the following new problems:

1. System-wide Identifier,
2. Node Creation,
3. Caller Interface,
4. Callee Interface.

7.1 System-wide Identifier

In a networking situation references consisting of an Ada access variable which correspond to a store address in the current machine are insufficient. They must be extended to a record holding a node identifier (machine plus program) and a store address in the identified machine. This presents no problems to the implementation.

7.2 Node Creation

Like a partition, a node has a create operation in its interface, which has parameters corresponding to the initialization parameters of the node instance being created. This is called from the body of the root procedure created in the Ada translation of the nodes. There is a small difference between the case of a distinguished node and any other node, in that the latter must return to caller, who invokes the create operation over the network, the (extended) reference to the node produced.