# Extensible Protected Types

O.P. Kiddle and A.J. Wellings
Department of Computer Science
University of York, YO1 5DD
U.K.
Email: andy@cs.york.ac.uk

## Abstract

This paper proposes extensions to Ada 95 to make protected types more object-oriented; in particular, we consider the notion of a tagged protected type. The semantic implications of this extension are considered along with how they can be implemented. Finally, we show two examples of extensible protected types and draw conclusions.

**Keywords**: concurrent object-oriented programming, Ada 95.

## 1 Introduction

Arguably, Ada 95 does not fully integrate its models of concurrency and object-oriented programming (Atkinson and Weller, 1993; Wellings et al., 1996; Burns and Wellings, 1998). For example, neither tasks nor protected objects are extensible. It is easy to argue that tasks are active components and, therefore, very different from Ada's tagged records. However, protected types would appear to be very similar to tagged types; at one level they simply require all their primitive operations to have mutual exclusion.

The nature of the Ada 95 development process was such that the extension to Ada 83 for object-oriented programming was, for the most part, considered separate to extensions for protected types. Although some consideration was given to abandoning protected types and instead using a Java-like synchronised methods in their place, there was no public debate of this issue. Similarly, there was no public debate on the issues associated with allowing protected types to be extended.

The purpose of this paper is to start a debate on what it would mean to allow protected types to be extended. In section 2, we briefly outline the various ways of achieving extensible synchronisation objects in concurrent object-oriented programming languages and place our work in context. In sections 3 and 4, we present a proposal for extensible protected types and discuss many of the associated semantic issues. In section 5, we consider how the proposed changes can be implemented, then in Section 6 we illustrate our approach with two examples. Finally, in section 7, we draw our conclusions.

## 2 Integrating Synchronisation and Object-Oriented Programming

In general, communication and synchronisation between concurrent processes require (Andrews and Schneider, 1983):

- mutual exclusive access to shared resources
- condition synchronisation to enable processes to coordinate their progress.

The approaches adopted by languages and operating systems can be classified (Liskov et al., 1986) as being either:

1. Avoidance synchronisation – where guards are placed on the operations requiring mutual exclusion to ensure that conditions are correct for the shared resource access. For example, Ada 95 uses guards on protected entries to express condition synchronisation.
2. Conditional wait – where operations requiring mutual exclusion have no guards but instead allow the calling process to be blocked inside the operation. For example, POSIX uses condition variables and mutexes to provide a monitor-like mechanism; mutexes provide the mutual exclusion and condition variables allow a thread to be blocked when the owner of a locked mutex is unable to proceed.

Within a concurrent object-oriented framework, there are several ways mutual exclusion and condition synchronisation can be handled:

1. methods in an object can be identified as requiring mutual exclusion; within the method conditional, waiting can be performed; this is the approach adopted by Java (Lea, 1997)
2. acceptance sets – a form of avoidance synchronisation where methods are executed in mutual exclusion and the object indicates which of its methods are "open" after each method invocation; for example, the ABCM language(Matsuoka and Yonezawa, 1993)
3. guarded methods – the traditional avoidance synchronisation method where boolean expressions are associated with methods; for example, the TAO language (Mitchell and Wellings, 1996)
4. explicit method acceptance where objects are active and are able to indicate explicitly when methods should be executed; for example, POOL-T(America, 1987).

Within this framework, we are investigating the traditional avoidance model, and whether Ada protected types can be made more object oriented.

## 3 A Proposal for Extensible Protected Types

The requirements for extensible protected types are easy to articulate. In particular, extensible (tagged) protected types should allow:

- new data fields to be added,
- new functions, procedures and entries to be added,
- functions, procedures and entries to be overridden,
- class-wide programming.

However, these simple requirements raise many semantic issues. We classify these issues as being either object-oriented issues or protected-object issues. During our discussions, we refer to our revised version of Ada as Ada-EPT (Ada with Extensible Protected Types).

### 3.1 Object-Oriented Issues of Extensible Protected Types

In this section we discuss several issues relating to the object-oriented aspects of our proposal. We focus on: primitive operations, dispatching, and redispatching.

#### 3.1.1 Primitive Operations

There are two possibilities for the definition of which operations are primitive on tagged protected types and hence can be dispatching:

1. Just those operations declared within the protected type. As protected types encapsulate operations, it would be natural to allow just these operations to be the primitive ones.

2. In addition to the operations declared directly within the protected type, other operations declared externally but within the same unit could be primitive. This would follow similar rules as for ordinary tagged types, so a parameter of the protected type would be needed for the external operation.

Consider the following example:

```
protected type T is tagged
  procedure X;
end T;

procedure Y(T1: in out T);
```

Under the first possibility, only the `X` procedure would be a primitive operation of `T`. A call to procedure `Y` would not be potentially dispatching.

The second possibility allows procedure `Y` also to be a primitive operation of `T`. However, clearly procedure `Y` would not be able to directly manipulate data in the instance `T1` of type `T` that it is passed, and would not have to obtain the lock for `T1` before executing. It would, of course, be able to call operations in the protected type.

Allowing external primitive operations would not add any additional expressive power to protected types but it could be useful. The advantage would be that the encompassing package could provide a more appropriate view of the protected type's functionality. Another use might be if there was a procedure that needed to dispatch on the protected type but did many other things not involved with the protected type itself. This could make it possible to minimise the time for which the protected object was locked and hence improve efficiency.

Our proposal allows both internal and external operations to be primitive.

#### 3.1.2 Dispatching

Consider the following example which is an adaptation of the example used by Barnes (1996) to describe the dispatching rules for ordinary tagged types:

```
protected type T is tagged
  procedure O(I: in Integer);
  procedure P(X: in out T);
  function F return T;
private
  ...
end T;

protected type T1 is new T with ... ;
-- inherits operations
```

Here, `O`, `P` and `F` are primitive operations of `T`. The actual parameter, `X` is a controlling operand and the result of a call to `F` is a controlling result. In addition, it is important to note that both `O`, `P` and `F` take an implicit parameter of type `T` which is also a controlling operand. All controlling operands and results of a call must be of the same type. Supposing we have the following variables:

```
A, B: T;
A1, B1: T1;
C: T'Class := ...; -- must be initialized
D: T'Class := ...;  -- as they are class wide
```

then, considering the simple case first, the following calls would be legal:

```
A.O(5);    -- non-dispatching, type T
C.O(5);    -- dispatching
```

Here, the first call does not dispatch because the type of `A` is known statically at compile time. The second call dispatches depending on the actual type of `C`. The following calls would also be legal:

```
A.P(B);    -- non-dispatching, type T
A1.P(B1);  -- non-dispatching, type T1
C.P(D);    -- dispatching
```

In the case of the first two calls, it is clear at compile time what the type of the controlling operands is, so the call can be determined statically. In the last call, a run-time check is performed to ensure that `C` and `D` are of the same type and a Constraint_Error is raised if not. Note, Ada 95 allows dispatching only on one parameter.

A call to `A.P(A1)` would be illegal because of the mixture of specific types. To avoid confusion, it is also illegal to mix static and dynamic types (as in a call to `A.P(C)`) although a view conversion could circumvent this restriction (and prevent any confusion).

A call to `C.P(D.F)` would be dispatching: the result of `D.F` would have the same tag as D so from the perspective of dispatching, it is similar to a call to `C.P(D)`.

External primitive operations would follow exactly the same rules as for ordinary tagged types. For example, consider the case where there is the following external procedure:

```
procedure Q(X: T; I: Integer);
```

Here, calls depend only on the X parameter, so a call to Q(A, 3) would be non-dispatching and of type T. A call to Q(C, 4) would be dispatching because the type of C is unknown at compile time.

An external primitive operation could be a function dispatching on its result. This creates an interesting scenario if this function is used as a parameter to a protected operation. If for example, there were the following functions external to the protected type:

```
function G return T;
function H(Y: T) return T;
```

then the results of G and H would be controlling results. Consider the following calls:

```
A.P(G);      -- non-dispatching, type T
C.P(G);      -- dispatching
C.P(H(D));   -- dispatching
A.P(H(G));   -- non-dispatching, type T
C.P(H(G));   -- dispatching
G.P(G);      -- illegal
```

As with ordinary tagged types, a function with a controlling result can adapt itself to the circumstances: in the first call, it needs to conform to the type of A so it uses type T. The second call is similar except that C is class wide so it has to dispatch to the actual type of C at run-time. With the third call, D and C must have conforming types at run-time and the H function will dispatch on this type. Note that it should also be possible to make a call to H(D).P(C). The fourth and fifth calls are similar to the first two but demonstrate that nesting of functions should work. Finally, the last call is illegal because it is ambiguous: there is no way to tell whether to deal with type T or T1.

An access to a tagged type can be a controlling operand so an access to a tagged protected type could also be a controlling operand.

### 3.1.3 Redispatching

In order for a primitive operation on an ordinary tagged type to be able to redispatch, it has to convert those operands on which redispatching is to occur into a class wide type using a view conversion and then call the primitive operation. Consider the following example:

```
type T is tagged record ...;

procedure P(X: T);
procedure Q(X: T) is
begin
  ...
  P(T'Class(X));  -- redispatch
  ...
end Q;

type T1 is new T with record ...;

procedure P(X: T1);

A1: T1;
```

Here, procedure Q does a redispatch. If Q was called with a parameter of T'Class(A1), it would initially dispatch to the Q procedure which takes a parameter of type T because this procedure is inherited by type T1. If Q then called P(X), it would be a direct call to the P procedure which takes a parameter of type T. As Q first converts its parameter (X) to a class wide type however, it would redispatch based on the actual type of X. If A1 had been passed to Q initially, it would therefore call the P procedure which takes a parameter of type T1.

Another issue is that an operation inside a tagged protected type, does not have the option of converting the object on which it originally dispatched to a class wide type because this object is passed implicitly to the operation. There are two possible strategies which can be taken:

1. make all calls to other operations from within a tagged protected type dispatching
2. use some form of syntactic change to make it possible to specify whether to redispatch

The first strategy is not ideal because it is often useful to be able to call an operation in the same type without redispatching. In addition, the first strategy is inconsistent with ordinary tagged types where redispatching is not automatic. We, therefore, favour the second strategy. Possible syntactic changes are discussed in section 4.3. For the same reasons, it is not possible for a method to call an overridden method in its parent type. This problem is also addressed by the syntactic changes.

### 3.1.4 Private Objects and Fields

One consideration is whether or not private fields can be seen in a derived type. In protected types, all data has to be declared as private so that it can not be changed without first obtaining mutual exclusion. There are four possible approaches to this:

1. Preventing a child protected object from accessing this private data. This would seriously limit the child's power to modify the behaviour of its parent object.
2. Allowing a child protected object full access to private data declared in its parent. This would be more powerful.
3. To provide more flexibility, additional keywords could be provided to distinguish between data that is fully private and data that is private but visible to child objects. This keyword would be used in a similar way to private. It might even be possible to distinguish between read-only and full access. The main problem of this approach is the additional keyword required.
4. An approach which would be more consistent with the way tagged types currently work in Ada would be to allow child protected objects to access private components of their parent protected type if they are declared in a child package of their parent protected type. This would however, be slightly inconsistent with the way protected types currently work in Ada because protected types do not rely on using packages to provide encapsulation.

We favour the second method because it provides flexibility and involves no new keywords.

## 3.2 Protected Object Aspect of Extensible Protected Types

In this section, we focus on the protected object aspects relating to extensible protected types. We consider procedures, functions and then issues associated with entries

### 3.2.1 Procedure Calls

If a procedure in a child protected type calls its parent, it should not have to wait to obtain the lock on the protected object before entering the parent procedure otherwise deadlock would occur. There is one lock for each instance of a protected type and the same lock should be used when the protected type is converted to a parent type. This is consistent with the current Ada approach when one procedure calls another in the same protected object.

### 3.2.2 Function Calls

A function call should be able to call either another function in the protected object or the function in its parent type which it overrides. As multiple functions (readers) are allowed to run concurrently, there is no reason why this would be prevented.

### 3.2.3 Entry Calls

There are a number of implications for entries that are primitive operations. The most notable implication is that for guards. If a child protected type overrides an entry from its parent, it should only be able to strengthen the guard (Meyer, 1997). This is because there is no way to deal with the situation where an entry in the child calls its parent entry and the parent's guard fails. If the parent's guard is evaluated with the child's then this problem cannot arise. Hence, a call to the parent entry is *not* considered potentially blocking.

If a tagged protected type is converted to its parent type using a view conversion external to the protected type and then an entry is called on that type, it is not clear which guard needs to be passed. There are three possible strategies that can be taken:

1. Prevent view conversions external to a protected object. This would severely limit the expressive power of the language.
2. Use the guard associated with the exact entry which is being called, ignoring any guard associated with an entry which overrides this exact entry. As the parent type does not know about new data added in the child, it could be argued that allowing an entry in the parent to execute when the child has strengthened the guard for that entry should be safe. This would mean that there would be an entry queue for each entry including separate entry queues for overridden entries as for those entries which override them.
3. Use the guard associated with the entry to which dispatching would occur if the object was converted to a class wide type (of the base object). This is the strongest guard and would allow safe re-dispatching in the entry body. This method results in one entry queue per entry instead of one for each entry and every overridden entry.

The second method is the most intuitive strategy but it does have a number of problems. One problem is that it would be unclear what the `'Count` attribute should return. There are three possible values:

1. the number of calls waiting on the exact entry specified
2. the number of calls waiting on the entry specified including any calls to entries which override that entry
3. the number of calls waiting on the entry specified including any calls to entries which override that entry or which that entry overrides

A step towards alleviating this problem could be achieved by using separate attributes for each value. This would be far from ideal because parent types should not have knowledge of the types which override them yet it is common for methods to call the method in their parent which they override. These calls might need accounting for.

Another problem with the second method is that due to using separate entry calls with different guards for overridden and overriding entries, it is harder to theorise about the order of entries being serviced. Normally entries are serviced in first-in, first-out (FIFO) order but with the separate queues, each with a separate guard, this might not be possible. For example, a later call to an overridden entry will be accepted before an earlier call to an overriding entry if the guard for the overridden entry becomes true with the overriding entry's guard remaining false.

The third approach is probably marginally more efficient due to the fewer entry queues. It does not suffer any problems from the issue of what the `'Count` attribute should return because it is treating the entries as one entry and `'Count` would accurately return the number of calls waiting on the guard. However, it violates Liskov and Wing (1994) substituability rule for objects. If the object is only ever treated as the parent type, then the entry will never become open. This is inevitable when synchronisation constraints are added to methods.

To demonstrate more clearly the differences between both semantic interpretations for entries, consider the following example:

```
protected type T is tagged
  entry E when E'Count > 1;
private
  I: Integer := 0;
end Parent;

protected type T1 is new T with
  entry E and when I > 0;
end Parent;

A: T1;
```

If a call was made to `A.E`, this would be statically defined as a call to `T1.E` and would be subject to its guard (`E'Count > 1 and I > 0`). The difficulty arises if a call was made to `T(A).E`. The view conversion here would make this a call to `T.E` but it is not immediately obvious which guard the call would wait for. For the purpose of this paper, we define that it would have to wait on `T1.E`'s guard (the third approach) for the reasons described earlier. In short, the tag of a type determines the guard for an entry call as opposed the entry to which it is dispatched.

### 3.2.4 Requeue

A requeue of an entry call to an entry within the same protected type could involve redispatching onto the correct operation in a similar manner as if the entry named in the requeue statement (the *target* entry) was called directly. It would also be possible that the requeue was to an inherited entry. This would mean that the *target* entry could well be in a different type. Redispatching would only occur when explicitly requested so for example, in a protected type T, `requeue E` would not be dispatching whereas `requeue T'Class.E` would be dispatching. Requeuing to a parent entry would require guard re-evaluation.

Requeues from other protected objects or accept statements could also involve dispatching to the correct operation in a similar way to if the target entry was called directly.

## 4 Syntax for Protected Extensible Types

The syntax for tagged protected types in Ada-EPT covers:

- protected type declaration
- guard declaration
- redispatching syntax

### 4.1 Protected Type Declaration

The syntax for tagged protected types would involve allowing declarations of the following form:

```
protected type SubClass is new SuperClass with
  . . .
end SubClass;
```

Written in the form used by the Ada Reference Manual (Intermetrics, 1995) this would change the syntax for a protected type declaration to:

```
protected_type_declaration ::=
    protected_new_declaration
  | protected_extension_declaration

protected_new_declaration ::=
    protected type defining_identifier
          [known_discriminant_part] is
      [[abstract] tagged]
        protected_definition;
      | private;

protected_extension_declaration ::=
    protected type defining_identifier
          [known_discriminant_part] is
      [abstract] new ancestor_subtype_indication with
        protected_definition;
      | private;
```

In short, the new syntax would allow a protected type as well as an ordinary record to be declared as tagged. It would not be possible to declare a protected object which was not declared as a type to be tagged.

No change would be required for the body of the protected type.

### 4.2 Guard Declaration

As was discussed earlier, if a child protected type overrides an entry from its parent, it would only be able to strengthen the guard. To facilitate this syntactically, `and when` could be used instead of just `when` when specifying the guard for the entry.

This results in the following syntax for an entry body declaration:

```
entry_body ::=
  entry defining_identifier
       entry_body_formal_part entry_barrier is
    declarative_part
  begin
    handled_sequence_of_statements
  end [entry_identifier];

entry_barrier ::= [and] when condition
```

### 4.3 Redispatching Syntax

As was discussed before, some form of syntactic change is required to allow an operation in a tagged protected type to redispatch. It is not possible to convert the protected object on which it is operating to a class wide type because it is an implicit parameter to the operation. We propose calls of the form `type.operation` should be allowed, where type is the type to convert the implicit protected object to

The following is an example of this syntax for a redispatch:

```
protected body T is
  ...
  procedure P is
  begin
    . . .
    T'Class.Q;
    . . .
  end P;
end T;
```

`T'Class` indicates the type to which the protected object (which is of type T'Class but which is being viewed as type T) that was passed to P implicitly should be view converted. This allows it to define which Q procedure to call. This syntax is also necessary to allow an operation to call an overridden operation in its parent.

This new syntax does not conflict with any other part of the language because it is strictly only a type that precedes the period. If it could be a protected object or an instance of a protected type then the call could be mis-interpreted as an external call: the Ada Reference Manual (Intermetrics, 1995) distinguishes between external and internal calls by the use, or not, of the full protected object name(Burns and Wellings, 1998). The call would then be a bounded error.

## 5 Implementing Extensible Protected Types

In general, there are two possible implementation strategies:

1. modify an Ada 95 compiler and run-time system
2. write a translator from Ada-EPT to Ada 95

Although, the GNAT is freely available in source form, the effort required to modify the code is significant.

The second implementation strategy is to write a conversion program to take Ada-EPT and convert it into Ada 95 which could then be compiled. We have investigated this possibility and considered using the Perl language to write such a translator. If a translator is to be written, the process could be made significantly simpler by providing much of the extra functionality required for tagged protected types in a separate Ada package which is linked with the program. This is very similar to the idea of a run-time system used by a compiler. This would result in the structure depicted by figure 1.

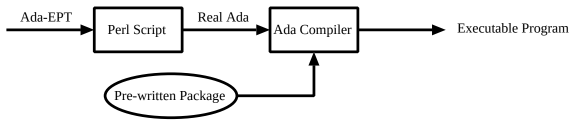

Figure 1: Structure of translator system

For the remainder of this paper we focus on the implementation of the pre-written package (called `Protection`).

## 5.1 Implementation Strategy

Burns and Wellings (1998) describe three strategies for providing synchronisation of data stored in tagged types. The first of these is to add synchronisation if and when it it is required by extending the object. This would be a complicated strategy to employ in a translator due to the difficult task of identifying where synchronisation is required. From a parent package this would be an impossible task.

The second strategy is to provide synchronisation in a base (root) object type. This is the strategy which we have chosen to investigate further because it allows much of the functionality to be demonstrated with an implementation of the base object and provides most of the features necessary for translation from Ada-EPT.

The third strategy is to use a protected type with the data passed as a discriminant. This strategy has the disadvantage that a descendant cannot add extra synchronised operations without using a rather complicated extension.

## 5.2 Protection Package

When showing how a synchronised object can be implemented by providing synchronisation in a base object, Burns and Wellings (1998) use a simple example where the base object contains one field of a protected *Mutex* type which only provides a lock and unlock mechanism. If the base object were to be usable by a translator from the Ada-EPT, the base object would need to provide more functionality than this to account for the need to provide more facilities than just mutual exclusion. The `Protection` package implements these requisite facilities which we will discuss in turn. The following is the specification for the `Protection` package:

```
with Ada.Finalization; use Ada.Finalization;
with Ada.Task_Identification;
use Ada.Task_Identification;

package Protection is

  subtype Guards is Positive;
  type Guard_Results is array
       (Guards range <>) of Boolean;

  type Protect_Base(Nog: Natural) is
       abstract tagged limited private;

  -- wait and done for use by functions
  procedure Wait_Read(Data: in out
                    Protect_Base'Class);
  procedure Done_Read(Data: in out
                    Protect_Base'Class);

  -- wait for use by procedures
  procedure Wait(Data: in out
               Protect_Base'Class);

  -- wait for use by entries
  procedure Wait(Data: in out
               Protect_Base'Class;
               Guard: Guards);

  -- signal for use by procedures or entries
  procedure Signal(Data: in out
                 Protect_Base'Class);

private

  protected type Lock(Nog: Natural) is
    entry Wait_Read;
    procedure Done_Read;
    entry Wait; -- for handling procedures
    entry Waitg(Guard: Guards);
    entry Unlock(Newg: Guard_Results);
  private
    entry Waitp; -- to hold waiting procedure calls
    entry Waite(1..Nog); -- for waiting on entries
    Locked: Natural := 1; -- Set to zero by unlock
    Lockingtask: Task_Id := Null_Task_Id;
    Readers: Boolean := False;
    G: Guard_Results(1..Nog);
  end Lock;

  type Protect_Base(Nog: Natural) is
       abstract new Limited_Controlled with
  record
    L: Lock(Nog);
  end record;

  function Reevaluate_Guards(Data: in Protect_Base)
          return Guard_Results;
  procedure Initialize(Object: in out Protect_Base);

end Protection;
```

### Support for Synchronisation

To provide synchronisation, the `Protection` package exports five procedures. There is a `wait` and `signal` procedure which allow a procedure to obtain exclusive access to the data. There is a second `wait` procedure to be used by entries. This `wait` procedure takes a parameter which allows the guard to be specified for the entry. The remaining two procedures (`Wait_Read` and `Done_Read`) are intended for functions and allow multiple readers (functions) to execute concurrently. Each of these five procedures merely call a corresponding procedure in the protected type (which is private) apart from `signal` which also re-evaluates the guards.

To implement the synchronisation, the `Protection` package uses two variables: one to count the number of readers and one to indicate if the object is currently locked. No reader can enter while the object is locked unless it is locked by another reader. A count of readers is kept so that the object can be unlocked when the last reader finishes. A writer can only enter when it has a true guard and the object is not locked unless it has already locked the object itself.

In order to satisfy this last condition whereby the same writer can lock an object multiple times, the `Ada.Task_Identification` package is used. When the object is locked, the `Task_Id` of the locking task is stored. This task is then passed through straight away otherwise deadlock would occur. This is a useful feature particularly as it is common in object-oriented programs for a method to call its corresponding method in the parent type. Note, however, currently the implementation in the `Protection` package does not conform to the Ada rules regarding protected objects. It would be possible for a call to be made to an external procedure which in turn calls back to the protected object. This should cause a bounded error.

One limitation in the way readers and writers are handled is that readers are given preference as a consequence of more than one being allowed to run concurrently. A number of simple modifications could be made to give preference to a write operation. For example, a reader could be prevented from entering if there was at least one waiting writer.

### Support for Object Orientation

To a certain extent, the `Protection` package (whose body is given in full in the Appendix) re-implements the facilities of a protected type and uses Ada's facilities for object orientation directly. Any tagged protected type in the Ada-EPT can be represented as a tagged type which is a child of the base object (`Protect_Base`) in Ada 95. Ada's normal facilities for tagged types such as the 'Class attribute, **abstract** and **private** keywords and view conversions can then be used.

As can be seen from the specification, the Protection package exports three types, two of which are purely for handling guards. The other is Protect_Base. Protect_Base has a field (and a discriminant). The field (L) is the protected type used for locking.

The Protect_Base type is actually a descendant of Ada.-Task_Identification.Limited_Controlled. This is to allow any guards to be evaluated based on the initial values of the object. The body of the Initialize package actually calls signal to avoid duplication of the code to evaluate the guards.

### Support for Entry Barriers

The method used for handling guards is complicated by the fact that there can be any number of them. The variable number of guards is dealt with by using a discriminant to Protect_Base which specifies the number of guards. The guards' values are then stored in an array which is bounded by the value of this discriminant. The disadvantage of this method is that child types need to be declared unbounded so that they can have further descendants. In addition, a subtype of the child needs to be defined which sets the restriction on number of guards and which is used for any instances of that type. For example, if there was a tagged protected type named Tagged_Protected which had two entries, the translated code would need the following declarations:

```
type Unc_Tagged_Protected is new
     Protection.Protect_Base with private;
subtype Tagged_Protected is Unc_Tagged_Protected(2);
```

The discriminant value would need to include any entries (guards) in a parent type.

The Protection package is responsible for manipulating the guards and determining which entry calls are given the *lock*. The values of the guards are however defined by the descendant type. To enable the Protection package to get hold of the guard values, a dispatching operation (Reevaluate_Guards) (which should be overridden by any descendant type that contains entries) is used.

### Entry Queue Handling

A family of entries handles the synchronisation associated with each Ada-EPT entry. The guard on these entries includes the actual guard value for each entry call so this package will deal with the entries in exactly the same manner as if they were genuine entries.

## 6 Example Usage

To demonstrate how the Protection package could be used, we take a couple of example problems written in the Ada-EPT and converted them manually into Ada 95 which makes use of the Protection package.

### 6.1 Simple Example

The first example is a simple protected type providing access to an item of shared data. This protected type contains one procedure to *Set* the value of the shared data and one function to *Get* the value.

The following is the Ada-EPT source to the problem:

```
generic
  type Item is private;
  Initial: Item;
package Read_Write is
  protected type Reader_Writer is tagged with
    procedure Set(Data: Item);
    function Get return Item;
  private
      Value: Item := Initial;
  end Reader_Writer;
end Read_Write;

package body Read_Write is
  protected body Reader_Writer is

    procedure Set(Data: Item) is
    begin
      Value := Data;
    end Set;

    function Get return Item is
    begin
      return Value;
    end Get;

  end Reader_Writer;
end Read_Write;
```

When converted to Ada 95, the above expands to the following:

```
with Protection; use Protection;
generic
  type Item is private;
  Initial: Item;
package Read_Write is
  type Unc_Reader_Writer is new
       Protection.Protect_Base with private;
  subtype Reader_Writer is Unc_Reader_Writer(0);

  procedure Set(Object: in out Unc_Reader_Writer;
                Data: Item);
  procedure Get(Object: in out Unc_Reader_Writer;
                Data: out Item);

private

  type Unc_Reader_Writer is new
       Protection.Protect_Base with record
    Value: Item := Initial;
  end record;

end Read_Write;

package body Read_Write is

  procedure Set(Object: in out Unc_Reader_Writer;
                Data: Item) is
  begin
    Protection.Wait(Object);
    Object.Value := Data;
    Protection.Signal(Object);
  exception
    when others =>
      -- Must always unlock when leaving procedure
      Protection.Signal(Object);
      raise;
  end Set;

  procedure Get(Object: in out Unc_Reader_Writer;
                Data: out Item) is
  begin
    Protection.Wait_Read(Object);
    Data := Object.Value;
    Protection.Done_Read(Object);
  exception
    when others =>
      Protection.Done_Read(Object);
      raise;
  end Get;

end Read_Write;
```

To allow further inheritance, what was a protected type, is now a tagged type. The two methods are primitive operations of this tagged type and have a parameter of its type (as opposed to this being implicit as with protected types). The function has had to be implemented as a procedure with an *out* parameter for the result because functions can only have *in* parameters and it is necessary for there to be write access to the tagged type in order to apply a read lock. The actual bodies of the methods are little changed except for the calls to `wait` and `signal` for the procedure and `Wait_Read` and `Done_Read` for the function at their start and end. Note also that it is necessary to trap all exceptions to ensure that the appropriate *unlock* method is called if the operation exits with an exception.

Creating a subtype of this package requires only to declare a descendant of `Unc_Reader_Writer`. The use of a generic package complicates the situation slightly because the package needs instantiating. The following is a package which also offers a `Compare` operation:

```
generic
package Read_Write_Comp is
  protected type Reader_Writer_Comp is
            new Reader_Writer with
    procedure Compare(Data: Item;
                      Result : out Boolean);
  end Reader_Writer_Comp;
end Read_Write_Comp;
```

This is implemented as:

```
with Read_Write;

generic
  type Item is private;
  Initial: Item;
package Read_Write_Comp is
  package Rw is new Read_Write(Item, Initial);
  use Rw;

  type Unc_Reader_Writer_Comp is
          new Rw.Unc_Reader_Writer with
      null record;
  subtype Reader_Writer_Comp is
          Unc_Reader_Writer_Comp(0);

  procedure Compare(X: in out Unc_Reader_Writer_Comp;
                    Y: in Item; Result: out Boolean);
end Read_Write_Comp;
```

The body of the package does not need to be any different as a result of not being a direct descendant of `protection`. The following is the body which merely defines the `Compare` procedure:

```
with Protection;

package body Read_Write_Comp is

  procedure Compare(X: in out Unc_Reader_Writer_Comp;
                    Y: in Item;
      Result: out Boolean) is
    Xi: Item;
  begin
    Get(X, Xi);
    Result := Xi = Y;
  end Compare;

end Read_Write_Comp;
```

Note that the procedure has to call `Get` to retrieve the value of protected data. This is because the value is in the private part of the `Read_Write` package and `Read_Write_Comp` could not be a child package and hence have access to the private data because of the use of generics. The `Compare` procedure does not need to call `Wait_Read` and `Done_Read` because it does not directly access the shared data and only makes one call to an operation which does. If such calls were used by `Compare`, it would not prevent the correct operation of the procedure. In fact, `Compare` could be an external primitive operation of the `Read_Write_Comp` tagged protected type.

## 6.2 Bounded Buffer

To demonstrate the handling of inheritance and entries, a second more complex problem is needed. The standard bounded buffer uses two entries so is appropriate. In order to demonstrate inheritance, a descendant of the bounded buffer is used which offers an extra method allowing a producer to get two items, forcing it to wait if only one is available. This could be useful if there were multiple consumers and one consumer needed to consume two consecutive items.

The following is the Ada-EPT source for the bounded buffer:

```
package Bounded_Buffer is

  Buffer_Size: constant Integer := 10;
  type Index is mod Buffer_Size;

  subtype Count is Natural range 0..Buffer_Size;
  type Buffer is array(Index) of Integer;

  protected type Bounded_Buffer is tagged
    entry Get(Item: out Integer);
    entry Put(Item: in Integer);
  private
    First: Index := Index'First;
    Last: Index := Index'Last;
    Number_In_Buffer: Count := 0;
    Buf: Buffer;
  end Bounded_Buffer;

end Bounded_Buffer;


package body Bounded_Buffer is

  protected body Bounded_Buffer is

    entry Get(Item: out Integer)
          when Number_In_Buffer > 0 is
    begin
      Item := Buf(First);
      First := First + 1; -- mod types cycle around
      Number_In_Buffer := Number_In_Buffer - 1;
    end Get;

    entry Put(Item: in Integer)
          when Number_In_Buffer < Buffer_Size is
    begin
      Last := Last + 1; -- mod types cycle around
      Buf(Last) := Item;
      Number_In_Buffer := Number_In_Buffer + 1;
    end Put;

  end Bounded_Buffer;

end Bounded_Buffer;
```

When converted to Ada 95, the above expands to the following:

```
with Protection; use Protection;

package Bounded_Buffer is

  Buffer_Size: constant Integer := 10;
  type Index is mod Buffer_Size;
  subtype Count is Natural range 0..Buffer_Size;
  type Buffer is array(Index) of Integer;


  -- Unconstrained and constrained versions of the
  -- type for inheritance/instances respectively
  type Unc_Bounded_Buffer is new
       Protection.Protect_Base with private;
  subtype Bounded_Buffer is Unc_Bounded_Buffer(2);

  procedure Get(Data: in out Unc_Bounded_Buffer;
                Item: out Integer);

  procedure Put(Data: in out Unc_Bounded_Buffer;
                Item: in Integer);

private

  Gdget: constant Guards := 1;
  Gdput: constant Guards := 2;

  type Unc_Bounded_Buffer is new
       Protection.Protect_Base with record
    First: Index := Index'First;
    Last: Index := Index'Last;
    Number_In_Buffer: Count := 0;
    Buf: Buffer;
  end record;

  function Reevaluate_Guards(Data: in Unc_Bounded_Buffer)
     return Guard_Results;

end Bounded_Buffer;


package body Bounded_Buffer is

  procedure Get(Data: in out Unc_Bounded_Buffer;
                Item: out Integer) is
  begin
    Protection.Wait(Data, Gdget);
    Item := Data.Buf(Data.First);
    Data.First := Data.First + 1;
    Data.Number_In_Buffer := Data.Number_In_Buffer - 1;
    Protection.Signal(Data);
  exception
    when others =>
      Protection.Done_Read(Object);
      raise;
  end Get;


  procedure Put(Data: in out Unc_Bounded_Buffer;
                Item: in Integer) is
  begin
    Protection.Wait(Data, Gdput);
    Data.Last := Data.Last + 1;
    Data.Buf(Data.Last) := Item;
    Data.Number_In_Buffer := Data.Number_In_Buffer + 1;
    Protection.Signal(Data);
  exception
    when others =>
      Protection.Done_Read(Object);
      raise;
  end Put;

  function Reevaluate_Guards(Data: in
    Unc_Bounded_Buffer) return Guard_Results is
  begin
    return (
      Gdget => Data.Number_In_Buffer > 0,
      Gdput => Data.Number_In_Buffer < Buffer_Size
    );
  end Reevaluate_Guards;

end Bounded_Buffer;
```

As can be seen from the source, entries are converted in a similar way as is the case for procedures with the exception that the guard is specified as a second parameter to `wait`. The other significant point is that `Reevaluate_Guards` has been overridden. It evaluates each of the guards and returns them in an array. There must be consistency with respect to which position in the array corresponds to which guard. In the example, constants are used to aid this but they are not essential.

The following is the Ada-EPT source for the descendant of a bounded buffer that offers a method for getting two data items:

```
with Bounded_Buffer; use Bounded_Buffer;

package Bounded_Buffer.Extended is

  protected type Extended_Buffer is
            new Bounded_Buffer with
    entry Get2(Item1, Item2: out Integer);
  end Extended_Buffer;

end Bounded_Buffer.Extended;


package body Bounded_Buffer.Extended is

  protected body Extended_Buffer is

    entry Get2(Item1, Item2: out Integer)
         when Number_In_Buffer > 1 is
    begin
      Item1 := Buf(First);
      Item2 := Buf(First+1);
      First := First + 2;
      Number_In_Buffer := Number_In_Buffer - 2;
    end Get2;

  end Extended_Buffer;

end Bounded_Buffer.Extended;
```

When converted to Ada 95, the above expands to the following:

```
with Bounded_Buffer; use Bounded_Buffer;

package Bounded_Buffer.Extended is

  type Unc_Extended_Buffer is
       new Unc_Bounded_Buffer with private;
  subtype Extended_Buffer is Unc_Extended_Buffer(3);

  procedure Get2(Data: in out Unc_Extended_Buffer;
      Item1, Item2: out Integer);

private
  Gdget2: constant Guards := 3;

  type Unc_Extended_Buffer is new
       Unc_Bounded_Buffer with null record;

  function Reevaluate_Guards(
      Data: in Unc_Extended_Buffer)
      return Guard_Results;

end Bounded_Buffer.Extended;

package body Bounded_Buffer.Extended is

  procedure Get2(Data: in out Unc_Extended_Buffer;
      Item1, Item2: out Integer) is
  begin
    Protection.Wait(Data, Gdget2);
    Item1 := Buf(First);
    Item2 := Buf(First+1);
    First := First + 2;
    Number_In_Buffer := Number_In_Buffer - 2;
    Protection.Signal(Data);
  exception
    when others =>
      Protection.Done_Read(Object);
      raise;
  end Get2;
```

```
    function Reevaluate_Guards(
        Data: in Unc_Extended_Buffer)
        return Guard_Results is
    begin
      return (
        Data.Number_In_Buffer > 0,           -- Get
        Data.Number_In_Buffer < Buffer_Size, -- Put
        Data.Number_In_Buffer > 1            -- Get2
      );
      -- Could have used a call to
      -- Reevaluate_Guards(Bounded_Buffer(Data))
      -- for the first two guards
    end Reevaluate_Guards;

  end Bounded_Buffer.Extended;
```

The Extended_Buffer type descends from Bounded_Buffer in the usual way for tagged types (except that the unconstrained versions are used as was discussed earlier) and so inherits all the methods for the bounded buffer. The new entry added to get two items follows exactly the same pattern as for those in the Bounded_Buffer package. Reevaluate_Guards again has to be overridden to describe the new guard. For reasons of clarity and efficiency, the guards for the inherited entries are explicitly defined but ideally, a view conversion should be used to convert the parameter to its parent type (Unc_Bounded_Buffer in this case) and the parent called for the inherited guards.

## 7 Conclusions

In this paper, we have considered the issue of introducing extensible protected types in Ada 95. The semantic model is fairly straight-forward with the exception of handling view conversions to protected objects when entries have been overridden and the guards strengthened.

It could be argued that as we have shown that extensible protected types can be implemented in Ada 95, there is no real need to modify the language. There are, however a few limitations to our solution which cannot be resolved using the existing Ada constructs alone. Furthermore, using the Protection package results in code which is far less readable than would be offered by the language extensions. In addition, the Ada 95 implementation is probably less efficient that could be achieved if the compiler directly supported the extensions.

## 8 Acknowledgements

The authors gratefully acknowledge the contributions of Pat Rogers, Alan Burns and Stuart Mitchell in the preparation of this paper.

This work has been undertaken as part of a final year project for the MEng degree in Computer Systems and Software Engineering at the University of York, U.K.

## APPENDIX

In this appendix, we present the full implementation of the Protection package.

```
  package body Protection is

    procedure Wait(Data: in out Protect_Base'Class) is
    begin
      Data.L.Wait;
    end Wait;
```

```
procedure Wait_Read(Data: in out Protect_Base'Class) is
begin
  -- wait for a function (functions can work concurrently)
  -- Note; in this model, functions are not
  -- prevented from modifying data
  Data.L.Wait_Read;
end Wait_Read;

procedure Done_Read(Data: in out Protect_Base'Class) is
begin
  Data.L.Done_Read;
end Done_Read;

procedure Wait(Data: in out Protect_Base'Class;
               Guard: Guards) is
begin
  Data.L.Waitg(Guard);
end Wait;

procedure Signal(Data: in out Protect_Base'Class) is
begin
  Data.L.Unlock(Reevaluate_Guards(Data));
end Signal;

function Reevaluate_Guards(Data: in Protect_Base)
         return Guard_Results is
begin
  return (1..0 => False);
  -- a null array; value is arbitrary
end Reevaluate_Guards;

procedure Initialize(Object: in out Protect_Base) is
begin
  Object.L.Unlock(Reevaluate_Guards(
                  Protect_Base'Class(Object)));
   -- Evaluate the guards and unlock the object
end Initialize;

protected body Lock is
  entry Wait_Read when Readers or Locked = 0 is
  begin
    Readers := True;
    Locked := Locked + 1;
  end Wait_Read;

  procedure Done_Read is
  begin
    Locked := Locked - 1;
    if Locked = 0 then Readers := False; end if;
  end Done_Read;

  entry Wait when True is
  begin
    -- wait for procedures,
    -- so they no need to pass an entry barrier
    if Lockingtask = Wait'Caller then
      -- pass if it is the locking task
      Locked := Locked + 1;
    else
      requeue Waitp;  -- wait until it is unlocked
    end if;
  end Wait;



  entry Waitp when Locked = 0 is
  begin
    -- wait for procedures therefore no barriers
    Locked := 1;
    Lockingtask := Waitp'Caller;
  end Waitp;

  entry Waitg(Guard: Guards) when True is
  begin
    if Lockingtask = Waitg'Caller then
      -- pass, if it is the locking task
      Locked := Locked + 1;
    else
       -- wait until it is unlocked
      requeue Waite(Guard);
    end if;
  end Waitg;
```

```
        entry Waite(for Q in 1..Nog) when
              G(Q) and Locked = 0 is
        begin
          Locked := 1;
          Lockingtask := Waite'Caller;
        end Waite;

        entry Unlock(Newg: Guard_Results) when
              Locked > 0 is
          -- Locked > 0 is really a pre-condition
          -- not a guard
        begin
          G := Newg;
          Locked := Locked - 1;
          if Locked = 0 then
            -- If this is the final unlock
            Lockingtask := Null_Task_Id;
          end if;
        end Unlock;

      end Lock;

    end Protection;
```

## References

America, P. (1987). POOL-T: A parallel object-oriented language, *Object-Oriented Concurrent Programming*, MIT Press, pp. 199–220.

Andrews, G. and Schneider, F. (1983). Concepts and notations for concurrent programming, *ACM Computing Surveys* **15**(1): 3–44.

Atkinson, C. and Weller, D. (1993). *Integrating Inheritance and Synchronisation in Ada9X*, Proceedings of TRI'Ada 93, ACM.

Barnes, J. (1996). *Programming in Ada 95*, Addison-Wesley.

Burns, A. and Wellings, A. J. (1998). *Concurrency in Ada*, second edn, Cambridge University Press.

Intermetrics (1995). Ada 95 reference manual, *ANSI/ISO/IEC-8652:1995*, Intermetrics.

Lea, D. (1997). *Concurrent Programming in Java*, Addison Wesley.

Liskov, B., Herlihy, M. and Gilbert, L. (1986). Limitations of remote procedure call and static process structure for distributed computing, *Proceedings of Thirteenth Annual ACM Symposium on Principles of Programming Languages*, St. Petersburg Beach, Florida, pp. 150–159.

Liskov, B. and Wing, J. (1994). A behavioral notion of subtyping, *ACM Transactions on Programming Languages and Systems* **16**(6): 1811–1841.

Matsuoka, S. and Yonezawa, A. (1993). Analysis of inheritance anomaly in object-oriented concurrent programming languages, *Research Directions in Concurrent Object-Oriented Programming*, MIT Press, pp. 107–150.

Meyer, B. (1997). *Object-Oriented Software Construction*, second edn, Prentice Hall.

Mitchell, S. E. and Wellings, A. J. (1996). Synchronisation, concurrent object-oriented programming and the inheritance anomaly, *Computer Languages* **22**(1).

Wellings, A. J., Mitchell, S. and Burns, A. (1996). Object-oriented programming with protected types in Ada 95, *International Journal of Mini and Micro Computers* **18**(3): 130–136.