Extensible Protected Types: Proposal Status

A.J. Wellings
Department of Computer Science, University of York, UK
Email: andy@cs.york.ac.uk

B. Johnson and B. Sanden
Department of Computer Science, Colorado Technical University, USA
Email: bjohnson@cos.colotechu.edu, bsanden@acm.org

J. Kienzle and T. Wolf
Software Engineering Laboratory
Swiss Federal Institute of Technology in Lausanne, Switzerland
Email: Joerg.Kienzle@epfl.ch, twolf@acm.org

S. Michell
Maurya Software, Ontario, Canada
Email: steve@maurya.on.ca

Abstract

At the last workshop, the issue of being able to extend protected types in the same manner as tagged types was discussed. The conclusion was that further consideration was needed before a definitive proposal could be made. The goal of this Position Paper is to summarise a full proposal and to identify the outstanding issues.

Keywords: concurrent object-oriented programming, inheritance anomaly, Ada 95.

1 Introduction

At the last workshop, the issue of being able to extend protected types in the same manner as tagged types was discussed [1]. The conclusion was that further consideration was needed before a definitive proposal could be made. After the workshop, a group of enthusiasts held numerous email discussions on how best to integrate Ada's object-oriented programming and protected type model. The results will shortly appear in the literature [5, 6]. The goal of this Position Paper is to summarise the proposal and to identify the outstanding issues. In section 2, we define the core extensions needed for our model. In section 3, we consider how these extensions can be integrated into the full language. Finally, in section 4 we summarise the outstanding issues are draw our conclusions. A full rational for our model along with some examples can be found in [4] (available at http://www.cs.york.ac.uk/ftpdir/reports/).

2 Extensible Protected Types – The Basic Model

2.1 Declaration and Primitive Operations

For consistency with the usage elsewhere in Ada, the word ‘tagged’ indicates that a protected type is extensible. A protected type encapsulates the operations that can be performed on its protected data. Consequently, the primitive operations of a tagged protected type are, in effect, already defined. They are, of course, similar to primitive operations of other tagged types in spirit but not in syntax, since other primitive operations are defined by being declared in the same package specification as a tagged type. Consider the following example:

```
protected type T is tagged
  procedure W (...);
  function X (...) return ...;
  entry Y (...);
private  -- data attributes of T
end T;

O : T;
```

`W`, `X`, and `Y` can be viewed as primitive operations on `T`. Interestingly, the call `O.X` takes a syntactic form similar to that in most object-oriented languages.

2.2 Inheritance

Tagged protected types can be extended in the same manner as tagged types:

```
protected type T1 is new T with
  procedure W (...); -- override T.W
  procedure Z (...); -- a new method
private -- new attributes of T1
end T1;
```

A child protected object has full access to private data declared in its parent.

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

2.3 Dispatching and re-dispatching

Given a hierarchy of tagged protected types, it is possible to create class-wide types and accesses to class-wide types; first consider only protected subprograms, for example:

```
type Pt is access protected type T'Class;
P: Pt := new . . .; -- some type in the hierarchy

P.W(...); -- dispatches to the appropriate projected object.
```

Of course from within `P.W`, it should be possible to convert back to the class-wide type and re-dispatch to another primitive operation. Unfortunately, an operation inside a tagged protected type does not have the option of converting the object (on which it was originally dispatched) to a class-wide type because this object is passed implicitly to the operation. We propose to use calls of the form `type.operation`, where `type` is the type to which the implicit protected object should be converted. The following is an example of this syntax for a re-dispatch:

```
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 in the hierarchy of type `T'Class` but which is being viewed as type `T`) that was passed implicitly to `P` 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, for example:

```
protected body T1 is -- an extension of T
  ...
  procedure W (...) is -- overrides the W procedure of T
  begin
    . . .
    T.W(...); -- calls the parent operation
    . . .
  end W;
end T1;
```

This new syntax does not conflict with any other part of the language because it is strictly only a type that precedes the period.

Requeuing can also lead to situations where re-dispatching is desirable. Requeuing to a parent entry would require barrier re-evaluation. Requeues from other protected objects or from accept statements in tasks could also involve dispatching to the correct operation in a similar way.

2.4 Entry Calls

Our proposal allows entries to be overridden with the constraint that child entries must strengthen their parent's barrier. The syntax **and when** is used to indicate this. To avoid having the body of a child protected object depend on the body of its parent, it is necessary to move the declaration of the barrier from the body to the specification of the protected type (private part). Consider

```
protected type T is tagged
  entry E ;
private
  I: Integer := 0;
  entry E when E'Count > 1; -- barrier given in the private part
end T;

protected type T1 is new T with
```

```
  entry E ;
private
  entry E and when I > 0;
end T;

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 barrier (`E'Count > 1 and I > 0`). The barrier would be repeated in the entry body.

There is only one queue for each overridden entry. In order to avoid inheritance anomalies[3]), we propose that all external calls to a protected object are dispatching. This is in conflict with regular tagged types, and to to resolve this, a new pragma could be introduced "External_Calls_Always_Dispatch" which would apply to regular tagged types.

2.4.1 Calling the Parent Entry and Parent Requeues

An integral part of OOP is the ability of a child object to call its parent's operations. The main problem is that Ada forbids an entry from explicitly calling another entry.

Our proposal is to allow the child entry to call the parent entry and for that call to be treated as a procedure call. It is clear that calling the parent entry is different from a normal entry call; special syntax has already been introduced to facilitate it (see section 2.3). As the parent call is viewed as a procedure call, it is not a potentially suspending operation and, therefore, allowed within a protected entry. However, the parent's barrier is still a potential cause for concern. One option is to view the barrier as an assertion and raise an exception if it is not true. [1] The other option is not to test the barrier at all, based on the premise that the barrier was true when the child was called and, therefore, need not be re-evaluated until the whole protected action is completed.

Even with this approach, there is still the problem that control is not returned to the child if the parent entry requeues requests to other entries for servicing. This, of course, could be made illegal and an exception raised. However, requeue is an essential part of the Ada 95 model and to effectively forbid its use with extensible protected types would be a severe restriction.

We propose that requeue in the parent is allowed. A consequence of this is that no post-processing is allowed after a parent call.

3 Integration into the Full Ada 95 Model

The above section has considered the basic extensible protected type model. Of course, any proposal for the introduction of such a facility must also consider the full implications of its introduction. This section considers the following topics:

- private types,
- abstract types, and

[1] Special consideration would need to be given to barriers which use the `'Count` attribute in the parent, since these will clearly change when the child begins execution.

- generics and mix-in inheritance

3.1 Private Types

Ada 95 supports the notion of private and limited private types. A protected type is a limited type, hence it is necessary to show how extensible protected types integrate into limited private types. The following illustrates how this is easily achieved.

In order to make a type private, its full definition is moved to the private part of the package. This can also be done for extensible protected types:

```
package Example1 is
  protected type Pt0 is tagged private;
private
  protected type Pt0 is tagged
    --  primitive operations.
    ...
  private
    --  data items etc.
    ...
  end Pt0;
end Example1;
```

Note that in this example, the primitive operations of type `Pt0` are all declared in the private part of the package and are thus visible only in child packages of package `Example1`. Other packages cannot do anything with type `Pt0`, because they do not have access to the type's primitive operations. Nevertheless, this construct can be useful for class-wide programming using access types, e.g. through

```
type Pt_Ref is access Pt0'Class;
```

Private types can also give a finer control over visibility. One might declare a type and make some of its primitive operations publicly visible while other primitive operations would be private (and thus visible only to child packages). For example:

```
package Example2 is
  protected type Pt1 is tagged
    -- public primitive operations, visible anywhere
    ...
  with private
    -- data items etc., see (1) below
    ...
  end Pt1;
private
  protected type Pt1 is tagged
    -- private primitive operations, visible only in child packages
    ...
  private
    -- additional data items etc., see (2) below
    ...
  end Pt1;
end Example2;
```

Note that the public declaration of type `Pt1` uses "with private" instead of only "private" to start its private section. This is supposed to give a syntactical indication that the public view of `Pt1` is an incomplete type that must be completed later on in the private part of the package.

The private parts of the incomplete and the full declaration of `Pt1` also have different visibility scopes:

1. The items declared in the private part of the public incomplete declaration are visible to types derived from `Pt1` anywhere.

2. The items declared in the private part of the full declaration of `Pt1` are visible to types derived from `Pt1` in child packages of package `Example2` only.

Extensible protected types thus offer even more visibility control than ordinary tagged types: the latter must declare all their data components either in the public or in the private part, whereas an extensible protected type may choose to make some of them public (to descendants only) and some of them private.

Alternatively a protected type can be declared to have a private extension. Given a protected type `Pt2`:

```
package Base is
  protected type Pt2 is tagged
    ...
  private
    ...
  end Pt2;
end Base;
```

A private extension can then be written as:

```
with Base;
package Example3 is
  protected type Pt3 is new Base.Pt2 with private;
private
  protected type Pt3 is new Base.Pt2 with
    --  Additional primitive operations
    ...
  private
    --  Additional data items
    ...
  end Pt3;
end Example3;
```

Here, only the features inherited from `Pt2` are publicly visible, the additional features introduced in the private part of the package are private and hence visible only in child packages of package `Example3`.

Private types can be used in Ada 95 to implement hidden and semi-hidden inheritance, two forms of implementation inheritance (as opposed to interface inheritance, i.e. subtyping). For instance, one may declare a tagged type publicly as a root type (i.e., not derived from any other type) while privately deriving it from another tagged type to reuse the latter's implementation. This hidden inheritance is also possible with extended protected types. Given the above package `Base`, hidden inheritance from `Pt2` can be implemented as follows:

```
with Base;
package Example4 is
  --  the public view of Pt4 is a root type
  protected type Pt4 is tagged
    --  primitive operations, visible anywhere
    ...
  with private
    --  data items etc.
    ...
  end Pt4;
private
```

```
  --  the private view of Pt4 is derived from Pt2
  protected type Pt4 is new Base.Pt2 with
    --  additional primitive operations, visible only in child packages
    ...
  with private
    --  additional data items etc.
    ...
  end Pt4;
end Example4;
```

The derivation of `Pt4` from `Pt2` is not publicly visible: operations and data items inherited from `Pt2` cannot be accessed by other packages. If some of the primitive operations inherited from `Pt2` should in fact be visible in the public view of `Pt4`, too, `Pt4` must re-declare them and implement them as call-throughs to the privately inherited primitive operations of `Pt2`. In child packages of package `Example4`, the derivation relationship is exposed and hence these inherited features are accessible in child packages.

Semi-hidden inheritance is similar in spirit, but exposes part of the inheritance relation. Given an existing hierarchy of extensible protected types:

```
package Example5_Base is
  protected type Pt5 is tagged
    ...
  private
    ...
  end Pt5;

  protected type Pt6 is new Pt5 with
    ...
  private
    ...
  end Pt6;
end Example5_Base;
```

One can now declare a new type `Pt7` that uses interface inheritance from `Pt5`, but implementation inheritance from some type derived from `Pt5`, e.g. from `Pt6`:

```
with Example5_Base; use Example5_Base;
package Example5 is
  protected type Pt7 is new Pt5 with
    ...
  with private
    ...
  end Pt7;
private

  protected type Pt7 is new Pt6 with
    ...
  private
    ...
  end Pt7;
end Example5;
```

As these examples show, extensible protected types offer the same expressive power concerning private types as ordinary tagged types. In fact, because protected types are an encapsulation unit in their own right (in addition to the encapsulation provided by packages), extensible protected types offer an even greater visibility control than ordinary tagged types. Primitive operations of an extensible protected type declared in the type's private section are visible only within that type itself or within a child extension of that type. Combining this kind of visibility (which is similar to Java's 'protected' declarator) with the visibility rules for packages gives some visibility specifications that do not exist for ordinary tagged types.

There is one difficulty with this scheme, though. It is currently possible in Ada 95 to define a limited private type that is implemented as a protected type. This raises the question whether the following should be legal:

```
package Example6 is
  type T is tagged limited private;
private
  protected type T is tagged
    ...
  private
    ...
  end T;
end Example6;
```

Here, although child packages could treat `T` as an extensible protected type, other client packages could do very little with the type. Furthermore, the mixture of protected and non-protected views of one and the same type may give rise to incalculable implementation problems because in some cases accesses to an object would have to be done under mutual exclusion even if the view of the object's type was not protected, simply because its full view was a protected type. Consequently, the kind of private completion shown in `Example6` is probably best disallowed.

3.2 Abstract Extensible Protected Types

Ada 95 allows tagged types and their primitive operations to be abstract. The Ada 95 model can easily be applied to extensible protected types. The following examples illustrate the integration:

```
protected type Ept is abstract tagged
   --  Concrete operations:
   function F (...) return ...;
   procedure P (...);
   entry E (...);

   --  Abstract operations:
   function F1 (...) return ... is abstract;
   procedure P1 (...) is abstract;
   entry E1 (...) is abstract;
private
   ...;
   entry E (...) when Cond;
end Ept;
```

The one issue that is perhaps not obvious concerns whether an abstract entry can have a barrier. On the one hand, an abstract entry cannot be called so any barrier is superfluous. On the other hand, the programmer may want to define an abstraction where it is appropriate to guard an abstract entry. For example:

```
protected type Lockable_Operation is abstract tagged
   procedure Lock;
   procedure Unlock;
   entry Operation (...) is abstract;
private
   Locked : Boolean := False;
   entry Operation (...) when not Locked;
end Lockable_Operation;
```

The bodies of `Lock` and `Unlock` set the `Locked` variable to the corresponding values. Now because of the barrier strengthening rule, the **when not** `Locked` barrier will automatically be enforced on any concrete implementation of the operation.

The above example can be rewritten with a concrete entry for `Operation` that has a null body. It should be noted, however, that with a concrete null-operation, one cannot force concrete children to supply an implementation for the entry. With an abstract entry, one can.

3.3 Generics and Mix-in Inheritance

Ada 95 does not support multiple inheritance. However, it does support various approaches which can be used to achieve the desired affect. One such approach is mix-in inheritance, which in Ada is done through generic packages that can take a parameter of a tagged type. The generic package provides the mixed-in components and operations; an instantiation then does the mix-in into an existing base type. A version of Ada with extensible protected types must also allow them to be parameters to generics and hence take part in mix-in inheritance.

As with normal tagged types, two kinds of generic formal parameters can be defined:

```
generic
   type Base_Type is [abstract] protected tagged private;
   type Derived_From is [abstract] new protected Derived [with private];
```

In the former, the generic body has no knowledge of the extensible protected type actual parameter. In the latter, the actual type must be a type in the tree of extensible protected types rooted at `Derived`.

Unfortunately, these facilities are not enough to cope with situations involving entries. Consider the case of a predefined lock which can be mixed in with any other protected object to define a lockable version. Without extra functionality, there is no way to express this. For these reasons, the generic modifier `entry <>` is used to mean all the entries of the actual parameter. The lockable mix-in type can now be achieved:

```
generic
   type Base_Type is [abstract] protected tagged private;
package Lockable_G is
   protected type Lockable_Type is new Base_Type with

      procedure Lock;
      procedure Unlock;
   private
      Locked : Boolean := False;
      entry <> and when not Locked;
   end Lockable_Type;
end Lockable_G;
```

The code **entry <> and when not** `Locked` indicates that all entries in the parent protected type should have their barriers strengthened by the boolean expression **not** `Locked`.

The **entry** <> feature makes it possible to modify the barriers of entries that are unknown at the time the generic unit is written. At the time the generic unit is instantiated, the entries of the actual generic parameter supplied for `Base_Type` are known, and **entry** <> then denotes a well-defined set of primitive operations.

This generic barrier modifier is similar to Frølund's "all-except" specifier [2], except that the latter also applies to primitive operations that are added later on in further derivations, whereas **entry** `<>` does not. If new primitive operations are added in further derivations, it is the programmer's responsibility to make sure that these new entries get the right barriers (i.e., include **when not** `Locked`).

4 Conclusions

This paper has summarised extensions to protected types to make them more object oriented. Inevitably we have had to make some compromises and there are still some outstanding issues. Perhaps the greatest compromise concerns the way in which a child entry can call its parent. We propose that this call be treated as an internal call and, therefore, not subject to blocking. However, given that an entry call can always requeue to another entry call, the child entry cannot assume that its parent call will return. This leads to a restriction of our model in that there can be no post processing of parent calls.

The outstanding issues raised by this work include:

- whether a call to a parent entry should check the guard and raise "Program Error" if it is not true;
- whether abstract entries should have guards;
- whether the facilities for mix-in inheritance are adequate for purpose.

References

[1] J.A. de la Puenta. New language features and other language issues. In *Proceedings of IRTAW9, Ada Letters, Vol XIX(2)*, pages 19–20, 1999.

[2] S. Fr\olund. Inheritance of synchronization constraints in cocurrent object-oriented programming languages. In *Proceedings of ECOOP '92, LNCS*, volume 615, pages 185–196. Springer, 1992.

[3] S. Matsuoka and A. Yonezawa. Analysis of inheritance anomaly in object-oriented concurrent programming languages. In *Research Directions in Concurrent Object-Oriented Programming*, pages 107–150. MIT Press, 1993.

[4] A.J. Wellings, B. Johnson, B. Sanden, J. Kienzle, T. Wolf, and S. Michell. Integrating object-oriented programming and protected types in Ada 95. YCS 316, Department of Computer Science, University of York, UK, 1999.

[5] A.J. Wellings, B. Johnson, B. Sanden, J. Kienzle, T. Wolf, and S. Michell. Integrating object-oriented programming and protected types in Ada 95. *ACM TOPLAS*, to appear.

[6] A.J. Wellings, B. Johnson, B. Sanden, J. Kienzle, T. Wolf, and S. Michell. Object-oriented programming and protected objects in ada 95. *Reliable Software Technologies - Ada-Europe 2000*, to appear.