Object-Oriented Programming and Protected Objects in Ada 95*

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

Integrating concurrent and object-oriented programming has been an active research topic since the late 1980s. The majority of approaches have taken a sequential object-oriented language and made it concurrent. A few approaches have taken a concurrent language and made it object-oriented. The most important of this latter class is the Ada 95 language which is an extension to the object-based concurrent programming language Ada 83. Arguably, Ada 95 does not fully integrate its models of concurrency and object-oriented programming. This paper discusses ways in which protected objects can be made more extensible.

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

1 Introduction

Arguably, Ada 95 does not fully integrate its models of concurrent and object-oriented programming [8, 2]. For example, neither tasks nor protected objects are extensible. When Ada 95 was designed, the extensions to Ada 83 for object-oriented programming were, for the most part, considered separate to extensions

*This paper extends and unifies the approaches described in [3] and [6].

to the concurrency model. Some consideration was given to abandoning protected types and instead using Java-like synchronised methods in their place, however, there was no public debate of this issue. Similarly, there was no public debate on the issues associated with allowing protected types or tasks to be extended, although a Language Study Note (LSN-1030) was produced which raised some of the issues discussed in this paper. The purpose of this paper is to discuss ways in which the Ada 95 concurrency model can be better integrated with object-oriented programming.

2 Concurrent Object-Oriented Programming

Integrating concurrent and object-oriented programming has been an active research topic since the late 1980s. The majority of approaches have taken a sequential object-oriented language and made it concurrent. A few approaches have taken a concurrent language and made it object-oriented. The most important of this latter class is the Ada 95 language which is an extension to the object-based concurrent programming language Ada 83.

In general, there are two main issues for concurrent object-oriented programming:

- the relationship between concurrent activities and objects – here the distinction is often between the concept of an active object, and where concurrent execution is created by the use of asynchronous method calls,
- the way in which concurrent activities communicate and synchronise (and yet avoid the so-called inheritance anomaly [4]).

3 The Ada 95 Programming Language

The Ada 83 language allowed programs to be constructed from several basic building blocks: packages, subprograms, and tasks. Of these, only tasks were considered to be types and integrated with the typing model of the language. Ada 95 extends the facilities of Ada 83 in areas of the language where weaknesses were perceived. Two of the main innovations were the introduction of data-oriented communication and synchronization through *protected types* and the introduction of object-oriented programming facilities through *tagged types*.

Although task types and protected types are fully integrated into the typing model of Ada 95, it is not possible to create a tagged protected type or a tagged task type. The designers shied away from this possibility partly because they felt that fully integrating object-oriented programming and concurrency was not a well-understood topic and, therefore, not suitable for an ISO standard professional programming language. Also, there were inevitable concerns that the scope of potential language changes being proposed was too large for the Ada community to accept. In spite of this, there is some level of integration between tagged types and tasks and protected objects. Tagged types after all are just part of the typing mechanism and therefore can be used by protected types and tasks types in the same way as other types. Indeed paradigms for their use have been developed (see [2] chapter 13). However, these approaches cannot get around the basic limitation that protected types and task types cannot be extended.

4 Making Concurrent Programming in Ada 95 more Object-Oriented

There are the following classes of basic types in Ada: scalar types - such as integer types, enumeration types, real types, etc, structured types - such as record types and array types, protected types, task types, and access types. Access types are special as they provide the mechanism by which pointers to the other types can be created. Note that, although access types to subprograms can be created, subprograms are not a basic type of the language.

In providing tagged types, Ada 95 has provided a mechanism whereby a structured type can be extended. It should be stressed, though, that only record types can be extended, not array types. This is understandable as the record is the primary mechanism for grouping together items which will represent the heterogeneous attributes of the objects. Furthermore, variable length array manipulation is already catered for in the language. Similarly, scalar types can already be extended using subtypes and derived types. Allowing records to be extended thus is consistent with allowing variable length arrays, subtypes and derived types.

A protected type is similar to a record in that it groups items together. (In the case of a protected type, these items must be accessed under mutual exclusion.) It would be consistent, then, to allow a protected type to be extended with additional items. The following sections will discuss some of the issues in allowing extensible protected types.

5 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 to be performed. These simple requirements raise many complex semantic issues.

5.1 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. Indeed, Ada's protected object syntax is in conflict with the language's usual representation of an 'object'.

5.2 Inheritance

This paper proposes that 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;
```

The issue of overriding protected entries will be considered in section 5.4.

One consideration is whether or not private fields in the parent type (T) can be seen in the child type (T1). 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 visibility issue:

1. Prevent a child protected object from accessing the parent's data. This would limit the child's power to modify the behaviour of its parent object, it only being allowed to invoke operations in its parent.

2. Allow a child protected object full access to private data declared in its parent. This would be more flexible but has the potential to compromise the parent abstraction.

3. Provide an additional keyword to distinguish between data that is fully private and data that is private but visible to child types. This keyword would be used in a similar way to `private` (much like C++ uses its keyword 'protected' to permit descendent classes direct access to inherited data items).

4. Allow child protected types to access private components of their parent protected type if they are declared in a child of the package in which their parent protected type is declared. This would be slightly inconsistent with the way protected types currently work in Ada because protected types do not rely on using packages to provide encapsulation.

This paper will assume the second method, as it provides the most flexibility, requires no new keywords and is consistent with normal tagged types.

If a procedure in a child protected type calls a procedure or function in its parent, it should 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.

5.3 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. There are two possible strategies: make all calls to other operations from within a tagged protected type dispatching, or use some form of syntactic change to make it possible to specify whether to re-dispatch or not. The first strategy is not ideal because it is often useful to be able to call an operation in the same type or a parent type without re-dispatching. In addition, the first strategy is inconsistent with ordinary tagged types where re-dispatching is not automatic. A solution according to the second strategy uses 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.

5.4 Entry Calls

Allowing entries to be primitive operations of extensible protected types raises many inter-related complex issues. These include:

1. Can a child entry call its parent's entry? – This is how reuse is achieved in an object-oriented framework. From the protected object perspective, however, calling an entry is a potentially suspending operation and these are not allowed within the body of a protected operation.

2. What is the relationship, if any, between the parent's barrier and the child's barrier? – There are three possibilities: no relationship, the child can weaken the parent's barrier, or the child can strengthen the parent's barrier.

3. How many queues does an implementation need to maintain for an overridden entry? – If there is no relationship between the parent and the child barrier, it is necessary to maintain a separate entry queue for each overridden entry. If there is more than one queue, the `'Count` attribute should reflect this. Hence `'Count` might give different values when called from the parent or when called from the child. A problem with using separate entry queues with different barriers for overridden and overriding entries is that it is harder to theorise about the order of entries being serviced. Normally entries are serviced in FIFO order but with separate queues, each with a separate barrier, this might not be possible.

4. What happens if a parent entry requeues to another entry? – When an entry call requeues to another entry, control is not returned to the calling entry but to the task which originally made the entry call. This means that when a child entry calls its parent and the parent entry requeues, control is not returned to the child. Given that the code of the parent is invisible to the child, this would effectively prohibit the child entry from undertaking any post-processing.

In order to reduce the number of options for discussion, it is assumed that child entries must strengthen their parent's barrier for the remainder of the paper. 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.

Even with barrier strengthening, the issue of barrier evaluation must be addressed. Consider the case where a tagged protected object 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 barrier needs to be passed. There are three possible strategies that can be taken:

1. Use the barrier associated with the exact entry which is being called, ignoring any barrier 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 barrier for that entry should be safe. Unfortunately, this is not the case. Consider a bounded buffer which has been extended so that the `Put` and `Get` operations can be locked. Here, if the lockable buffer is viewed converted to a normal buffer and `Get/Put` called with only the buffer barriers evaluated, a locked buffer will be accessible even if it is locked. Furthermore, this approach would also mean that there would be separate entry queues for overridden entries.

2. Use the barrier associated with the entry to which dispatching would occur if the object was converted to a class wide type (i.e., the barrier of the entry of the object's actual type). This is the strongest barrier 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. However, it is perhaps misleading as it is the parent's code which is executed but the child's barrier expression that is evaluated.

3. Allow view conversions from inside the protected object but require that all external calls are dispatching calls. Hence, there is only one entry queue, and all external calls would always invoke the primitive operations of the object's actual type. The problem with this approach is that currently Ada does not dispatch by default. Consequently, this approach would introduce an inconsistency between the way tagged types and extensible protected types are treated.

For the remainder of this paper, it is assumed that external calls to protected objects always dispatch.

5.4.1 Calling the Parent Entry and Parent Requeues

So far this section has discussed the various issues associated with overridden entry calls. However, details of how the child entry actually calls its parent have been left unspecified. The main problem is that Ada forbids an entry from explicitly calling another entry. There are two approaches to this problem.

1. Use requeue. – Although Ada forbids nested entry calls, it does allow an entry call to be requeued. Hence, the child can only requeue to the parent. Requeue gives the impression of calling the parent but it is not possible for the child to do any post-processing once the parent entry has executed (as the call returns to the caller of the child entry). As a requeue, the parent's barrier would have to be re-evaluated. Given that the child barrier has strengthened the parent's barrier, the parent's barrier would normally be open. If this is not the case, an exception is raised (to queue the call would require more than one entry queue). Furthermore, if atomicity is to be maintained and the parent requeue is to be part of the same protected action, the parent entry must be serviced before any other entries whose barriers also happen to be open. Hence, this requeue has slightly different semantics from a requeue between unrelated entries.

2. Allow the child entry to call the parent entry and treat that call 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 5.3). In this approach, the parent call is viewed as a procedure call and therefore not a potentially suspending operation. 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. * 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.

With either of these approaches, 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.

The remainder of this paper will assume a model where parent calls are treated as procedure calls (the issue of the assertion is left open) and requeue in the parent is allowed. A consequence of this is that no post-processing is allowed after a parent call.

6 Inheritance Anomaly

The combination of the object-oriented paradigm with mechanisms for concurrent programming may give rise to the so-called "inheritance anomaly" [4]. An inheritance anomaly exists if the synchronization between operations of a class is not local but may depend on the whole set of operations present for the class. When a subclass adds new operations, it may therefore become necessary to change the synchronization defined in the parent class to account for these new operations. This section examines how extensible protected types can deal with this inheritance anomaly.

Synchronization for extensible protected types is done via entry barriers. An entry barrier can be interpreted in two slightly different ways:

- as a precondition (which must become a guard when concurrency is introduced in an object-oriented programming language, as [5] argues). In this sense, entries are the equivalent of partial operations.
- as a synchronization constraint.

The use of entry barriers (i.e., guards) for synchronization makes extended protected types immune against one of the kinds of inheritance anomalies identified by [4]: guards are not subject to inheritance anomalies caused by a partitioning of states.

To avoid a major break of encapsulation, it is mandatory for a concurrent object-oriented programming language to have a way to re-use existing synchronization code defined for a parent class and to incrementally modify this inherited synchronization in a child class. In our proposal, this is given by the **and when** clause, which incrementally modifies an inherited entry barrier and hence the inherited synchronization code.

*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.

Inheritance anomalies in Ada 95 with extended protected types can still occur, though. As [7] argue, the root cause of inheritance anomalies lies in a lack of expressive power of concurrent object-oriented programming languages: if not all five criteria identified by [1] are fulfiled, inheritance anomalies may occur. Ada 95 satisfies only three of these criteria; synchronization based on history information cannot be expressed directly using entry barriers (local state must instead be used to record execution history), and synchronization based on request parameter values also is not possible directly in Ada 95. The example for the resource controller shown in section 7 exhibits both of these inheritance anomalies. Because the barrier of entry `Allocate_N` cannot depend on the parameter `N` itself, an internal requeue to `Wait_For_N` must be used instead. The synchronization constraint for `Wait_For_N` itself is history-sensitive: the operation should be allowed only after a call to `Deallocate` has freed some resources. As a result, `Deallocate` must be overridden to record this history information in local state, although both the synchronization constraints for `Deallocate` itself as well as its functionality remain unchanged.

7 Example – Advanced Resource Control

Resource allocation is a fundamental problem in all aspects of concurrent programming. Its consideration exercises all Bloom's criteria (see section 6) and forms an appropriate basis for assessing the synchronisation mechanisms of concurrent languages, such as Ada.

Consider the problem of constructing a resource controller that allocates some resource to a group of client agents. There are a bounded number of instances of the resource; contention is possible and must be catered for in the design of the program. [7] propose the following resource controller problem as a benchmark for concurrent object-oriented programming languages.

Implement a resource controller with 4 operations:

- `Allocate`: to allocate one resource,
- `Deallocate`: to deallocate a resource
- `Hold`: to inhibit allocation until a call to
- `Resume`: which allows allocation again.

There are the following constraints on these operations:

1. `Allocate` is accepted when resources are available and the controller is not held (synchronization on local state and history)
2. `Deallocate` is accepted when resources have been allocated (synchronization on local state)
3. calls to `Hold` must be serviced before calls to `Allocate` (synchronization on type of request)
4. calls to `Resume` are accepted only when the controller is held (synchronization on history information).

As Ada 95 has no deontic logic operators, not all history information can be expressed directly in barriers. However, it is possible to use local state variables to record execution history. The following solution simplifies the presentation by modelling the resources by a counter indicating the number of free resources. Requirement 2 is interpreted as meaning that an exception can be raised if an attempt is made to deallocate resources which have not yet been allocated.

```
package Rsc_Controller is
  Max_Resources_Available : constant Natural := 100; --  For example
  No_Resources_Allocated : exception; -- raised by deallocate

  protected type Simple_Resource_Controller is tagged
    entry Allocate;
    procedure Deallocate;
    entry Hold;
    entry Resume;
  private
    Free   : Natural := Max_Resources_Available;
    Taken  : Natural := 0;
    Locked : Boolean := False;
    entry Allocate    when Free > 0 and not Locked and   --  req. 1
                           Hold'Count = 0;               --  req. 3
    entry Hold        when not Locked;
    entry Resume      when Locked;                       --  req. 4
  end Simple_Resource_Controller;
end Rsc_Controller;
```

The body of this package simply keeps track of the resources taken and freed, and sets and resets the `Locked` variable.

```
package body Rsc_Controller is

 protected body Simple_Resource_Controller is
    entry Allocate when Free > 0 and not Locked and Hold'Count = 0 is
    begin ... end Allocate;
    procedure Deallocate is begin ... end Deallocate;
    entry Hold when not Locked is begin ... end Hold;
    entry Resume when Locked is begin ... end Resume;
  end Simple_Resource_Controller;
end Rsc_Controller
```

[7] then extend the problem to consider the impact of inheritance:

> Extend this resource controller to add a method: `Allocate_N` which takes an integer parameter `N` and then allocates `N` resources. The extension is subject to the following additional requirements:
>
> 5. Calls to `Allocate_N` are accepted only when there are at least `N` available resources.
> 6. Calls to `Deallocate` must be serviced before calls to `Allocate` or `Allocate_N`.

Note that this specification is flawed, and the implementation shown in [7] also exhibits this flaw: if `Deallocate` is called when no resources are allocated, the resource controller will deadlock and not service any calls to `Deallocate`, `Allocate`, or `Allocate_N`. In this implementation, this has been corrected implicitly, because calling `Deallocate` when no resources are allocated is viewed as an error and an exception is raised.

Requirement 5 is implemented by requeuing to `Wait_For_N` if not enough resources are available.

Requirement 6 is implicitly fulfiled because calls to `Deallocate` are never queued since `Deallocate` is implemented as a procedure.

```
with Rsc_Controller; use Rsc_Controller;
package Advanced_Controller is

  protected type Advanced_Resource_Controller is
    new Simple_Resource_Controller with
    entry Allocate_N (N : in Natural);
    procedure Deallocate;
      --  Ada-specific anomaly: because barriers cannot access parameters,
      --  we must also override this method so that we can set 'Changed'.
  private
     entry Allocate_N when Free > 0 and not Locked and     --  req. 1
             Hold'Count = 0;                               --  req. 3

    Current_Queue : Boolean := False;
      --  Indicates which of the two 'Wait_For_N' entry queues is the one
      --  that currently shall be used. (Two queues are used: one queue
      --  is used when trying to satisfy requests, requests that cannot
      --  be satisfied are requeued to the other. Then, the roles of the
      --  two queues are swapped. This avoids problems when the calling
      --  tasks have different priorities.)

    Changed       : Boolean := False;
     --  Set whenever something is deallocated. Needed for correct
     --  implementation of 'Allocate_N' and 'Wait_For_N'. Reset each
     --  time outstanding calls to these routines have been serviced.
     --  'Changed' actually encodes the history information "Wait_For_N"
     --   and is only accepted after a call to 'Deallocate'.

    entry Wait_For_N (for Queue in Boolean) (N : in Natural);
    entry Wait_For_N (for Queue in Boolean) when
         not Locked and Hold'Count = 0 and (Queue = Current_Queue) and Changed;
      --  This private entry is used by 'Allocate_N' to requeue to if
      --  less than N resources are currently available.
   end Advanced_Resource_Controller;
end Advanced_Controller;

package body Advanced_Controller

  protected body Advanced_Resource_Controller is
    procedure Deallocate is
      --  Overridden to account for new history information encoding
      --  needed for access to parameter in the barrier of Allocate_N.
    begin ... end Deallocate;

    entry Allocate_N (N : in Natural) when Free > 0 and
              not Locked and Hold'Count = 0 is
    begin ... end Allocate_N;

    entry Wait_For_N (for Queue in Boolean)(N : in Natural) when
               not Locked and Hold'Count = 0 and
              (Queue = Current_Queue) and Changed is
    begin ... end Wait_For_N;
  end Advanced_Resource_Controller;
end Advanced_Controller;
```

8 Conclusions

This paper has argued that Ada 95's model of concurrency is not well integrated with its object-oriented model. It has focussed on the issue of how to make protected types extensible and yet avoid the pitfalls of the inheritance anomaly. The approach adopted has been to introduce the notion of a tagged protected type which has the same underlying philosophy as normal tagged types.

Although the requirements for extensible protected types are easily articulated, there are many potential solutions. The paper has explored the major issues and, where appropriate, has made concrete proposals. Ada is an extremely expressive language with many orthogonal features. The paper has shown that the introduction of extensible protected types does not undermine that orthogonality. The work presented here, however, has not been without its difficulties. The major one is associated with overridden entries.

The paper 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. It has been beyond the scope of this paper to address these issues - see [9].

References

[1] T. Bloom. Evaluating synchronisation mechanisms. In *Proceedings of the Seventh ACM Symposium on Operating System Principles*, pages 24–32, Pacific Grove, 1979.

[2] A. Burns and A. J. Wellings. *Concurrency in Ada.* Cambridge University Press, second edition, 1998.

[3] O. P. Kiddle and A. J. Wellings. Extended protected types. In *Proceedings of ACM SIGAda Annual International Conference (SIGAda 98)*, pages 229–239, November 1998.

[4] 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.

[5] B. Meyer. *Object-Oriented Software Construction.* Prentice Hall, second edition, 1997.

[6] S. Michell and K. Lundqvist. Extendible dispatchable task communication mechanisms. In *Proceedings of IRTAW9, Ada Letters, Vol XIX(2)*, pages 54–59, 1999.

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

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

[9] 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.