Integrating Object-Oriented Programming and Protected Objects in Ada 95

A. J. WELLINGS
University of York

B. JOHNSON and B. SANDEN
Colorado Technical University

J. KIENZLE and T. WOLF
Swiss Federal Institute of Technology in Lausanne
and
S. MICHELL
Maurya Software

Integrating concurrent and object-oriented programming has been an active research topic since the late 1980's. There is now a plethora of methods for achieving this integration. 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. For example, neither tasks nor protected objects are extensible. This article discusses ways in which protected objects can be made more extensible.

Categories and Subject Descriptors: D.3.3 [**Programming Languages**]: Language Constructs and Features—*Concurrent programming structures and inheritance*

General Terms: Languages
Additional Key Words and Phrases: Concurrency, concurrent object-oriented programming, inheritance anomaly, Ada 95

This article extends and unifies the approaches described in Kiddle and Wellings [1998], Michell and Lundqvist [1999], and Johnson [2000].

Author's addresses: A. J. Wellings, Dept. of Computer Science, University of York Heslington, York, YO10 5DD, UK; B. Johnson and B. Sanden, Colorado Technical University, 4435 N. Chestnut Street, Colorado Springs, CO 80907; J. Kienzle and T. Wolf, Software Engineering Laboratory, Swiss Federal Institute of Technology in Lausanne CH – 1015 Lausanne EPFL, Switzerland; S. Michell, Maurya Software, 29 Maurya Court, Ottawa, Ontario, Canada, K1G5S3.

1. INTRODUCTION

Ada 95 is the only international standard programming language that supports object-oriented real-time distributed systems. However, it has been argued [Atkinson and Weller 1993; Wellings et al. 1996; Burns and Wellings 1998] that the language does not have a well-integrated set of facilities for concurrent object-oriented programming. The object-oriented mechanisms are built around the concept of tagged types and take their inspiration from Oberon's type extensibility model [Wirth 1988]. Unfortunately, neither tasks types (the unit of concurrency) nor protected types (essentially monitors) are extensible.

The purpose of this article is to discuss ways in which the Ada 95 concurrency model can be better integrated with its object-oriented programming facilities. The article is structured as follows. Section 2 introduces the main problems associated with the integration of object-oriented and concurrent programming. Section 3 then describes the main features of the Ada 95 language that are relevant to this work. Section 4 argues that Ada 95 does not have a well-integrated object-oriented concurrency model. To achieve better integration, Section 5 proposes that Ada's protected type mechanism be made extensible and discusses the main syntactic and semantic issues. Section 6 then considers how extensible protected types integrate with Ada's general model of abstraction and inheritance. Sections 7 and 8 discuss how the proposals address the inheritance anomaly and how they can be used in conjunction with the current object-oriented mechanisms. Section 9 presents some extended examples, and Section 10 draws conclusions from this work.

2. CONCURRENT OBJECT-ORIENTED PROGRAMMING

Integrating concurrent and object-oriented programming has been an active research topic since the late 1980's. There is now a plethora of methods for achieving this integration (see Wyatt et al. [1992] or Briot[1998] for a review). The majority of approaches have taken a sequential object-oriented language and made it concurrent (for example, the various versions of concurrent Eiffel [Meyer 1993; Caromel 1993; Karaorman and Bruno 1993]). 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. A full discussion of this language will be given in the next section.

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 (which by definition will execute concurrently with other active objects, for example Maio et al[1989], Mitchell and Wellings [1996], and Newman [1998]) and where concurrent execution is created by the use of asynchronous method calls (or early returns from method calls) [Yonezawa et al. 1986; Yokote and Tororo 1987; Corradi and Leonardi 1990]

—the way in which concurrent activities communicate and synchronize (and yet avoid the so-called inheritance anomaly [Matsuoka and Yonezawa 1993]): see Mitchell and Wellings [1996] for a summary of the various proposals.

Perhaps the most interesting recent development in concurrent object-oriented programming is Java [Lea 1997; Oaks and Wong 1997]. Here we have, notionally, a

new language which is able to design a concurrency model within an object-oriented framework without worrying about backward compatibility issues. The Java model integrates concurrency into the object-oriented framework by a form of active objects. All descendants of the predefined class `Thread` have the predefined methods `run` and `start`. When `start` is called, a new thread is created, which executes `run`. Subclassing `Thread` and overriding the `run` method allows an application to express active objects. (It is also possible to obtain `run` by implementing the interface `Runnable`.) Other methods available on the `Thread` class allow for a wide range of thread control. Communication and synchronization are achieved by allowing any method of any object to be specified as "synchronized". Synchronized methods execute with a mutual exclusion lock associated with the object. All classes in Java are derived from the `Object` class that has methods which implement a simple form of condition synchronization. A thread can, therefore, wait for notification of a single event. When used in conjunction with synchronized methods, the language provides the functionality similar to that of a simple monitor [Hoare 1974].

Arguably, Java provides an elegant, although simplistic, model of object-oriented concurrency.

3. THE ADA 95 PROGRAMMING LANGUAGE

The Ada 83 language allowed programs to be constructed from several basic building blocks: packages, subprograms (procedures and functions), and tasks. Of these, only tasks were considered to be types and integrated with the typing model of the language. Just as with any other type in Ada; many instances of a task type can be declared; tasks can be placed in arrays and records, and pointers to tasks can be declared and created. Tasks can encapsulate data objects as well as other tasks. They communicate synchronously through entries and provide capabilities to control that communication by selection and acceptance of entry calls. In conclusion, Ada 83 fully integrated its concurrency model into the sequential components of the language. They are built using a consistent underlying type model.

3.1 Data-Oriented Synchronization: Protected Types

Ada 95 extends the facilities of Ada 83 in areas of the language where weaknesses were perceived. One of the innovations was the introduction of data-oriented communication and synchronization through *protected types*.

Instances of a protected type are called protected objects; they are basically monitors [Hoare 1974] but avoid the disadvantages associated with the use of low-level condition variables. Instead, protected types may have guarded entries similar to those provided by conditional critical regions [Brinch-Hansen 1972].

A protected type in Ada 95 encapsulates some data items, which can only be accessed through the protected type's operations. It is declared as shown in the following example:

```
protected type Shared_Int is
  --  Public operations
  procedure Set (Val : in Integer);
  function  Get return Integer;
  entry     Wait_Until_Zero;
private
```

```
  --  Encapsulated data
  Current : Integer := 0;
  --  Private operations might follow here
end Shared_Int;
```

The operations of this protected type are implemented in a corresponding body:

```
protected body Shared_Int is
  procedure Set (Val : in Integer) is
  begin
    Current := Value;
  end Set;

  function  Get return Integer is
  begin
    return Current;
  end Get;

  entry     Wait_Until_Zero
     when Current = 0 is     -- Entry barrier (guard)
  begin
     null;
  end Wait_Until_Zero;
end Shared_Int;
```

Instances of this protected type, i.e., protected objects, can be declared just like any other variable:

```
X : Shared_Int;  --  A protected object named 'X'
```

Operations on this shared object can be invoked in the following way:

```
X.Set(42);
Some_Variable := X.Get;
X.Wait_Until_Zero;
```

Calls to the operations of a protected type are so-called *protected actions* and guarantee mutually exclusive access to a protected object with the usual semantics of multiple readers (function calls, which are read-only) or one writer (procedure and entry calls).

When an entry is called, and its barrier is false, the call is queued, and the calling task is blocked until the call has been finally executed. Otherwise, the call is accepted and executed in a protected action. At the end of each procedure or entry call, the barriers of all entries are examined. If a barrier has become true, a possibly queued call is then executed as part of the same protected action, i.e., without relinquishing the mutual exclusion in between. This servicing of entry queues is repeated until either there are no more queued calls or until all their barriers are false. The protected action then terminates.

The following example illustrates the use of entries with a simple bounded buffer, where items can only be taken from the buffer when it is not empty, and items can be put into it only when it is not full.

```
protected type Integer_Bounded_Buffer is
  entry Put (I : in  Integer);
  entry Get (I : out Integer);
private
```

```
  Buffer      : array (1 .. 10) of Integer;
  First, Last : Natural := 1;
  Nof_Items   : Natural := 0;
end Integer_Bounded_Buffer;

protected body Integer_Bounded_Buffer is
  entry Put (I : in  Integer)
     when Nof_Items < Buffer'Length is
  begin
    Buffer (Last) := I;
    Last := Last mod Buffer'Length + 1;
    Nof_Items := Nof_Items + 1;
  end Put;
  entry Get (I : out Integer)
     when Nof_Items > 0 is
  begin
    I := Buffer(First);
    First := First mod Buffer'Length + 1;
    Nof_Items := Nof_Items - 1;
  end Get;
end Integer_Bounded_Buffer;
```

If `Get` is called when `Nof_Items` is zero, the caller is queued. When another task calls `Put`, `Nof_Items` will be incremented. When the entry queues are serviced after the call to `Put` has finished, the barrier of `Get` is now true, and the queued call is allowed to proceed, thus unblocking the task that made that call.

A requeue statement of the form

```
requeue Target_Entry;
```

allows an entry to put a call, which it has already begun processing, back on the same or some other entry queue. A requeue immediately leaves the current entry, requeues the call, and then initiates entry queue servicing. Once the requeued call has been executed, control is returned to the task that made the original call. A caller is typically requeued if, after consulting the parameters, it is found that the request could not be immediately met. Requeue is also used when a caller must be made to wait for the result of a request. For example, a protected entry may issue a hardware command then requeue the caller until an interrupt arrives indicating that the command has been performed. An example of the requeue statement can be found in Section 9.2.

Within the operations of a protected type, the attribute `E'Count` represents the number of calls in the queue of entry `E`.

Potentially blocking calls, in particular entry calls, are forbidden within a protected action. This language rule helps avoid deadlocks due to the nested monitors problem and avoids a possible unbounded priority inversion that might otherwise occur. This means that a procedure of a protected type may call other procedures or functions of the same or some other protected object, but not entries. Functions of a protected type may only call other protected functions of the same protected object to avoid circumventing the read-only restriction. However, they may call both protected functions and procedures of other protected objects. Entries may call procedures or functions, but not other entries; they may only requeue to another entry.

3.2 Object-Orientation: Tagged Types

One of the other main extensions to Ada 83 was the introduction of object-oriented programming facilities. Here the designers of Ada 95 were faced with a dilemma. Ada 83's facility for encapsulation was the package. Unfortunately, packages (unlike tasks) were not fully integrated into the typing model: there were no package types. Rather than introduce a class-like construct into the language (as had been done by almost all other object-oriented languages), Ada 95 followed the Oberon [Wirth 1988] approach and achieved object-orientation by type extension. The designers argued that Ada 83 already had the ability to derive types from other types and override their operations. Consequently, object-orientation was achieved via the introduction of "tagged types".

Tagged types in Ada 95 are record types that can be extended. Thus a class in Ada is represented by the following:

```
package Objects is
  type Class is tagged record
    -- data attributes of the class
  end record;

  -- the following are the primitive operations of the type
  procedure Method1 (O: in Class; Params: Some_Type);
  procedure Method2 (O: in out Class; Params : Some_Type);
end Objects;
```

The data attributes of the class in the above example are directly visible to users of the class. Ada 95 also allows these attributes to be fully encapsulated by using private types:

```
package Objects is
  type Class is tagged private;

  -- the following are the primitive operations of the type
  procedure Method1 (O: in Class; Params: Some_Type);
  procedure Method2 (O: in out Class; Params : Some_Type);
private
  type Class is tagged record
    -- data attributes of the class
  end record;
end Objects;
```

Objects of the class can be created and used by

```
with Objects; use Objects;
...
Object: Class;
Params: Some_Type;

....
begin
  Method1(Object, Params);
end;
```

Contrast this to a call to an object's method in the more typical object-oriented paradigm where the call is of the form: `Object.Method1(Params)`. The difference is purely syntactical; both forms have the same expressive power and denote the

same language construct, namely, a call to a primitive operation of a tagged type or a call of a method of a class, respectively.

Inheritance in Ada 95 is achieved by extending the parent type and overriding the primitive operations.

```
with Objects; use Objects;
package Extended_Objects is
  type Extended_Class is new Class with
    -- new data attributes
  end record;

  -- overridden primitive operations
  procedure Method1 (O: in Extended_Class; Params: Some_Type);
  procedure Method2 (O: in out Extended_Class; Params : Some_Type);

  -- new primitive operation
  procedure Method3 (O: in out Extended_Class; Params : Some_Type);
end Extended_Objects;
```

Polymorphism in Ada 95 is achieved by the use of class-wide types or pointers to class-wide types. It is possible, for example, to declare a pointer to a hierarchy of tagged types rooted at a place in the tree of type extensions. This pointer can then reference any object in the type hierarchy. When a primitive method is called passing the dereferenced pointer, run-time dispatching occurs to the correct operation:

```
type Pointer is access Object.Class'Class;
  -- 'Class indicates a class-wide type

Ap: Pointer := new ...; -- some object derived from Object.Class;

...

Method1(Ap.all, Param);  -- dispatches to appropriate method
```

In Ada 95, dispatching only occurs when the actual parameter of a call to a primitive operation is of a class-wide type. This contrasts with some other object-oriented programming languages where dispatching is the default (e.g., Java). In order to force dispatching in Ada, the parameter must be explicitly converted to a class-wide type when invoking the primitive operation. This situation often occurs when one primitive operation of an object wants to dispatch to some other primitive operation of the same object. This is called redispatching and can be achieved by converting the operand to a class-wide type, as shown in the following example:

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

procedure P (X: T) is ...;
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, by explicitly converting the parameter X to a class-wide type before invoking P. If this conversion had been omitted and Q just called P(X), then the call would be statically bound to the procedure P of T, regardless of what actual parameter was passed to Q.

It should be noted that Ada allows calls to overridden operations to be statically bound from outside the defining tagged type. For example, although the Extended_Objects package (defined earlier) has extended the Class tagged type and overridden Method1, it is possible for a client to write:

```
Eo: Extended_Class;
 ...
Method1(Class(Eo), ...);
```

and call the overridden method explicitly. Arguably this has now broken the Extended_Class abstraction, and perhaps should be disallowed. Such explicit conversions can only be safely done from within the overridden method itself when it wishes to call its parent method.

3.3 Child Packages

Child packages are another extension to Ada 83. Their main motivation is to add more flexibility to the single-level packaging facility. With Ada 83, changes to a package which resulted in modifications to the specification required recompilation of all clients using that package. This is at odds with object-oriented programming which facilitates incremental changes. Furthermore, extending private tagged types is not feasible without further language additions, as access to data in private types can only be made from within the package body.

Consider the following example given in the previous section:

```
package Objects is
  type Class is tagged private;

  -- the following are the primitive operations of the type
  procedure Method1 (O: in Class; Params: Some_Type);
  procedure Method2 (O: in out Class; Params : Some_Type);
private
  type Class is tagged record
    -- data attributes of the class
  end record;
end Objects;
```

To extend this class and have visibility of the parent data attributes, would require the package to be edited.

A child package has direct access to the private sections of its parents and grandparents without going through their interfaces. Hence, we have

```
package Objects.Extended_Objects is
  -- "." indicates that package Extended_Objects is a child of Objects

  type Extended_Class is new Class with private;
```

```
    -- overridden primitive operations
    procedure Method1 (O: in Extended_Class; Params: Some_Type);
    procedure Method2 (O: in out Extended_Class; Params : Some_Type);

    -- new primitive operation
    procedure Method3 (O: in out Extended_Class; Params : Some_Type);
  private

    type Extended_Class is new Class with private;
      -- new data attributes
    end record;
  end Extended_Objects;
```

allows the implementation of the new and overridden primitive operations to have access to the original class's data attributes.

3.4 Object-Oriented Programming and Concurrency

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 Burns and Wellings[1998], chapter 13). However, these approaches cannot get around the basic limitation that protected types and task types cannot be extended.

4. MAKING ADA 95 CONCURRENT PROGRAMMING MORE OBJECT-ORIENTED

Now that the dust is beginning to settle around the Ada 95 standard, it is important to begin to look to the future. The object-oriented paradigm has largely been welcomed by the Ada community. Even the real-time community, which was originally sceptical of the facilities and worried about the impact they would have on predictability, is beginning to see some of the advantages. Furthermore, as people become more proficient in the use of the language, they begin to realize that better integration between the concurrency and object-oriented features would be beneficial. The goal of this article is to continue the debate on how best to achieve full integration in any future version of the language.

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

—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 (procedures and functions) can be created, subprograms are not a basic type of the language.

With tagged types, Ada 95 provides 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. The issues associated with extensible task types are the subject of on-going research.

5. EXTENSIBLE PROTECTED TYPES

To make protected types more integrated with the object-oriented programming model requires modifications to the Ada 95 syntax and semantics. The modifications center around the notion of an extensible (tagged) protected type. The requirements for extensible protected types are easy to articulate. In particular, they should allow

—new data fields to be added,

—new functions, procedures, and entries to be added,

—functions, procedures, and entries to be overridden, and

—class-wide programming to be performed.

These simple requirements raise many complex semantic issues. Furthermore, any proposed extensions should be fully integrated with the Ada model of object-oriented programming.

5.1 Declaration and Primitive Operations

For consistency with the usage elsewhere in Ada, the word "tagged" indicates that a protected type is extensible. As described in Section 3.1, 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 -- new proposed syntax
  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" (see Section 3.2).

5.2 Inheritance

Tagged protected types can be extended in the same manner as tagged types. Hence,

```
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 have to be declared as private so that they cannot 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 behavior 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 are fully private and data that are 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.

The remainder of this article will assume the second method, as it provides the most flexibility and requires no new keywords. It is also 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 the same lock should be used when the protected object is converted to a parent type. This is consistent with the current Ada approach when one procedure/function calls another in the same protected object.

5.3 Dispatching and Redispatching

Given a hierarchy of tagged protected types, it is possible to create class-wide types and access types to class-wide types, e.g.,

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

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

Of course from within `P.W`, it should be possible to convert back to the class-wide type and redispatch 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 which can be taken:

(1) make all calls to other operations from within a tagged protected type dispatching or
(2) use some form of syntactic change to make it possible to specify whether to redispatch 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 redispatching. In addition, the first strategy is inconsistent with ordinary tagged types where redispatching is not automatic.

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 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 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, e.g.,

```
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. If it could be an instance of a protected type then the call could be misinterpreted as an external call: the Ada Reference Manual [Taft and Duff 1997] 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.

Requeuing can also lead to situations where redispatching is desirable. Just as with procedures, redispatching would only occur when explicitly requested, so, for example, in a protected type `T`, `requeue E` would not dispatch whereas `requeue T'Class.E` would. Requeuing to a parent entry would require barrier reevaluation. 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 interrelated complex issues. These include:

(1) *Can a child entry call its parent's entry?* From an object-oriented perspective, it is essential to allow the child entry to call its parent. This is how reuse is achieved. From the protected-object perspective, calling an entry is a potentially suspending operation, and these are not allowed within the body of a protected operation (see Section 3.1). It is clear that a compromise is required and that a child entry must be able to extend the facilities provided by its parent.

(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. Frølund [1992] suggests that as the child method extends the parent's method, the child must have more restrictive synchronization constraints, in order to ensure that the parent's state remains consistent.[1] However, he also indicates that if the behavior of the child method totally redefines that of the parent, it should be possible to redefine the synchronization constraints. Alternatively, it can also be argued that the synchronization constraints of the child should weaken those of the parent, not strengthen them, in order to avoid violating the substitutability property of subtypes [Liskov and Wing 1994].

(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

[1]Where the child has access to its parent's state, barrier strengthening is not a sufficient condition to ensure the consistency of that state, as the child can make the barrier false before calling the entry. See also the discussion in Section 5.4.1.

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 theorize about the order of entries being serviced. Normally, entries are serviced in first-in, first-out (FIFO) order, but with separate queues, each with a separate barrier, 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 barrier for the overridden entry becomes true with the overriding entry's barrier remaining false.

(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 (see Section 3.1). 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 postprocessing.

In order to reduce the number of options for discussion, for the remainder of the article it is assumed that child entries must strengthen their parent's barrier. The syntax **and when** is used to indicate this.[2] 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 then 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:

[2]It is assumed that **and when** is a short-circuit control form.

(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 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. The problems associated with maintaining more than one entry queue per overridden entry have already been mentioned.

(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 redispatching in the entry body. This method results in only one entry queue per entry instead of one for each entry and one for 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 article, it is assumed that external calls to protected objects always dispatch.[3]

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 (see Section 3.1). There are several approaches to this problem.

(1) *Use requeue.* Although Ada forbids nested entry calls, it does allow entry requeuing. Hence, the child entry can requeue to the parent. After the parent entry has executed, control returns to the caller of the child entry, however, so the child entry cannot do any postprocessing. As a part of the requeue, the parent's barrier is evaluated. It should normally be open given that the child barrier has strengthened it; if not, an exception is raised. (To queue the call would require more than one entry queue.)[4] Furthermore, if the child and parent entries are to form one atomic protected action, the parent entry must

[3]To harmonize with regular tagged types a new pragma could be introduced called "External_Calls_Always_Dispatch" which would apply to regular tagged types.

[4]With the requeue approach and multiple entry queues, there need not be any relationship between the parent and the child barriers. Such an approach has already been ruled out in the previous subsection.

be serviced before any other entries whose barriers happen to be open. Hence, this requeue has slightly different semantics than 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.[5] 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 reevaluated 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 article 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 postprocessing is allowed after a parent call.

6. 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

—generics and mix-in inheritance

6.1 Private Types

The encapsulation mechanism of Ada 95, the package, gives the programmer great control over the visibility of the entities declared in a package. In particular, Ada 95 supports the notion of private and limited private types, i.e., types whose internal structure is hidden for clients of the packages (where the types are declared) and that can be modified only through the primitive operations declared in these packages (for these 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:

[5]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.

```
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), e.g.,

```
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 semihidden 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 redeclare 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.

Semihidden 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.

6.2 Abstract Extensible Protected Types

Ada 95 allows tagged types and their primitive operations to be abstract. This means that instances of the type cannot be created. An abstract type can be an extension of another abstract type. A concrete tagged type can be an extension from an abstract type. An abstract primitive operation can only be declared for

an abstract type. However, an abstract type can have non abstract primitive operations.

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, e.g.,

```
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.

6.3 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 [Frølund 1992], 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`).

7. INHERITANCE ANOMALY

The combination of the object-oriented paradigm with mechanisms for concurrent programming may give rise to the so-called "inheritance anomaly" [Matsuoka and Yonezawa 1993]. 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 Meyer [1997] argues). In this sense, entries are the equivalent of partial operations [Herlihy and Wing 1994].

—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 Matsuoka and Yonezawa [1993]: 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 reuse 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.

Inheritance anomalies in Ada 95 with extended protected types can still occur, though. Bloom [1979] suggested that the application programmers need to be able to express the synchronization between processes according to the following constraints (Blooms original analysis was in the context of a client server model):

(1) *The type of request*: The server might wish to accept requests in an order which is determined by the type of request message. In object-oriented terms, the type of a message is the method which is to be invoked in the called object. Therefore given an object with, say, methods A, B, and C, the server might wish to execute method A in preference to B and B in preference to C, etc.

(2) *The order of request*: A server might wish to service requests in FIFO, priority of caller, or non deterministic order. In the object model, this requires the methods to be executed according to order of the method invocation.

(3) *Request Parameters*: The arguments of the request often dictate whether a message can or cannot be accepted. For example, a method `get(n)` to obtain `n` items from a buffer can only be accepted if `n` items are available. Hence in an object model it may be necessary to block certain method calls according to the value of their parameters.

(4) *Local State*: A server might not be in a position to synchronize with a client if it is in a certain state, for example when a bounded buffer is empty or full.

In the object model, this synchronization is based on information contained in instance variables of the object.

(5) *History Information*: This is synchronization based on whether a given request (message invocation, in the object model) has occurred. This is often closely related to *local state*, since past executions may well have changed the state; however, it is sometimes convenient to maintain it as a separate category, as it may be easier to express certain constraints this way.

As Mitchell and Wellings [1996] 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 Bloom are fulfilled, 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 9.2 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.

The **entry <>** modifier has been introduced in Section 6.3 to allow protected objects created using mix-in inheritance to affect the barriers of their parent. In the `Lockable_G` example presented in Section 6.3, all the barriers are strengthened by adding the condition `not Locked`. It may well be that the inherited procedures need to be similarly guarded. This gives rise to an Ada-specific inheritance anomaly. As synchronization is done via barriers, only entries can be synchronized, but not procedures. If the synchronization constraints of a subtype should restrict an inherited primitive operation that was implemented as a procedure in the parent type, the subtype would have to override this procedure by an entry. However, when using class-wide programming, a task may assume that a protected operation is implemented as a procedure (as that is what the base type indicates) and is therefore non blocking. At run-time the call might dispatch to an entry and block on the barrier, which would make the call illegal if it occurred within a protected action. For these reasons, overriding procedures with entries should not be allowed for extensible protected types.

As discussed in Section 6.3, further Ada-specific inheritance anomalies that might arise when mix-in inheritance is used can be avoided by providing additional functionality for generics. Because the generic mix-in class must define the synchronization for the *complete class* resulting from the combination of the mix-in class with some a priori unknown base class, the **entry <>** barrier modifier was introduced. It allows the mix-in class to impose its own synchronization constraints on an unknown set of inherited operations. However, the new generic barrier modifier **entry <>** alone is not sufficient to avoid the introduction of new Ada-specific inheritance anomalies. It is also necessary to have a way for the mix-in class to adapt the

synchronization of its additional primitive operations to the synchronization constraints imposed by an actual base type. When the generic mix-in is instantiated with some base type to create a new result type, it must be possible to parameterize the mix-in's synchronization based upon the base type in order to obtain the correct synchronization for the new result type. How such a parameterization could be obtained is still a topic of on-going research.

8. INTERACTION WITH TAGGED TYPES

So far, the discussion has focused on how protected types can be extended. This section now considers the interaction between tagged types and protected tagged types.

Consider the following which defines a simple buffer:

```
package Simple_Buffer is
  type Data_T is tagged private;
  procedure Write (M : in out Data_T; X : Integer);
  procedure Read  (M : in Data_T;     X : out Integer);
private
   type Data_T is tagged
     record
       I : Integer := 4;
     end record; -- say
end Simple_Buffer;
```

Such a buffer can only be used safely in a sequential environment. To make a prewritten buffer safe for concurrent access requires it to be encapsulated in a protected type. The following illustrates how this can easily be achieved.

```
protected type Buffer is tagged
  procedure Write (X : Integer);
  procedure Read  (X : out Integer);
private
  D : Simple_Buffer.Data_T;
end Buffer;
```

The buffer can now only be accessed through its protected interface.

Of course if the `Buffer` protected type is extended, the following will dispatch on the buffer.

```
type B is access Buffer'Class;

Buf : B := new ...;


Buf.Write(3);
```

Alternatively, `Simple_Buffer.Data_T` can be made protected but not encapsulated by the following:

```
protected type Buffer is tagged
  procedure Write (M : in out Simple_Buffer.Data_T; X : Integer);
  procedure Read  (M : in out Single_Data_T;        X : out Integer);
private
  ..;
end Buffer;
```

This would allow the buffer to be accessed directly (without the protection overheads) where the situation dictates that it is safe to do so.

Combining extensible protected types with class-wide tagged types allows for even more powerful paradigms. Consider

```
protected type Buffer is tagged
  procedure Write (M : in out Simple_Buffer.Data_T'Class;
                   X : Integer);
  procedure Read  (M : in out Single_Data_T'Class;
                   X : out Integer);
private
 ..;
end Buffer;
```

Here, both the protected type and the tagged type can be easily extended. The program can arrange for dispatching on the `Buffer` and from within the `Write/Read` routines. Further, by using access discriminants the data can be encapsulated and protected from any concurrent use.

```
type Ad is access Simple_Buffer.Data_T'Class;

protected type Buffer(A : Ad) is tagged  -- a normal discriminant
  procedure Write (X : Integer);
  procedure Read  (X : out Integer);
private
 ...
end Buffer;

type B is access Buffer'Class;

B1 : B := new Buffer(new Simple_Buffer.Data_T)...;
```

Here, `B1` will dispatch to the correct buffer, and Write/Read will dispatch to the correct data which will be encapsulated.

9. EXAMPLES

This section presents two examples illustrating the principles discussed in this article. They assume all external calls dispatch; there is no postprocessing after parent calls, and no checking of parents' barriers; and they assume that the child has access to the parent's state.

9.1 Signals

In concurrent programming, signals are often used to inform tasks that events have occurred. Signals often have different forms: there are transient and persistent signals, those that wake up only a single task, and those that wake up all tasks. This section illustrates how these abstractions can be built using extensible protected types.

Consider first, an abstract definition of a signal.

```
package Signals is

  protected type Signal is abstract
    procedure Send;
    entry Wait is abstract;
```

```
  private
    Signal_Arrived : Boolean := True;
  end Signal;

  type All_Signals is access Signal'Class;
end Signals;

package body Signals is

  protected body Signal is abstract
    procedure Send is
    begin
      Signal_Arrived := True;
    end Send;
  end Signal;
end Signals;
```

Now to create a persistent signal:

```
with Signals; use Signals;
package Persistent_Signals is

  protected type Persistent_Signal is new Signal with
    entry Wait;
  private
    entry Wait when Signal_Arrived;
  end Persistent_Signal;
end Persistent_Signals;

package body Persistent_Signals is

  protected body type Persistent_Signal is
    entry Wait when Signal_Arrived is
    begin
      Signal_Arrived := False;
    end;
  end Persistent_Signal;
end Persistent_Signals;
```

To create a transient signal

```
with Signals; use Signals;
package Transient_Signals is

  protected type Transient_Signal is new Signal with
    procedure Send;
    entry Wait;
  private
    entry Wait when Signal_Arrived;
  end Transient_Signal;
end Transient_Signals;

package body Transient_Signals is

  protected body type Transient_Signal is
    procedure Send is
    begin
      if Wait'Count = 0 then
```

```
        return;
      end if;
      Signal.Send;
    end Send;

    entry Wait when Signal_Arrived is
    begin
      Signal_Arrived := False;
    end;
  end Transient_Signal;
end Transient_Signals;
```

To create a signal which will release all tasks.

```
generic
  type Base_Signal is new protected Signal;
package Release_All_Signals is

  protected type Release_All_Signal is new Base_Signal with
    entry Wait;
  private
    entry Wait and when True;
  end Release_All_Signal;
end Release_All_Signals;

package body Release_All_Signals is

  protected body Release_All_Signal

    entry Wait and when True is
    begin
      if Wait'Count /= 0 then
        return;
      end if;
      Base_Signal.Wait;
    end;
  end Release_All_Signal;
end Release_All_Signals;
```

Now, of course,

```
My_Signal : All_Signals := ...;

My_Signal.Send;
```

will dispatch to the appropriate signal handler.

9.2 Advanced Resource Control

Resource allocation is a fundamental problem in all aspects of concurrent programming. Its consideration exercises all Bloom's criteria (see Section 7) and forms an appropriate basis for assessing the synchronization 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 number of instances of the resource, but the number is bounded; contention is possible and must be catered for in the design of the program. Mitchell and Wellings [1996] 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 (which thus becomes available again for allocation)
> —`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).

In Ada 95, 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 modeling 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. Hence, it is represented by a protected procedure rather than an entry.

```
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
      Free := Free -1; -- allocate resource
      Taken := Taken + 1;
    end Allocate;

    procedure Deallocate is
    begin
      if Taken = 0 then
         raise No_Resources_Allocated;
      end if;
      Free := Free + 1; -- return resource
      Taken := Taken - 1;
    end Deallocate;

    entry Hold       when not Locked is
    begin
      Locked := True;
    end Hold;

    entry Resume     when Locked is
    begin
      Locked := False;
    end Resume;
  end Simple_Resource_Controller;
end Rsc_Controller
```

Mitchell and Wellings [1996] 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. (Synchronization on request parameters.)
> 6. Calls to `Deallocate` must be serviced before calls to `Allocate` or `Allocate_N`. (Synchronization on the type of request.)

The additional constraint that calls must be serviced in a FIFO_Within_Priorities fashion is ignored here. Mitchell and Wellings [1996] also do not implement this, and in Ada 95, it would be done through pragmas.

Note that this specification is flawed, and the implementation shown in Mitchell and Wellings [1996] 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' (see below).

 private
  entry Allocate_N when
       Free > 0 and not Locked and          -- req. 1
       Hold'Count = 0;                -- req. 3
   -- Note: Ada does not allow access to parameters in a barrier
   -- (purely for efficiency reasons). Such cases must in Ada
   -- always be implemented by using internal suspension of the
   -- method through a requeue statement. Everything below is just
   -- necessary overhead in Ada 95 to implement the equivalent of
   -- having access to parameters in barriers.

  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'
   -- is only accepted after a call to 'Deallocate'.

  entry Wait_For_N (for Queue in Boolean) (N : in Natural);
   -- This declares two entries with names "Wait_For_N (True)"
   -- and "Wait_For_N (False)". 'Allocate_N' requeues to one of the
   -- entries if less than N resources are currently available.
   -- Two entries are required to ensure correct behavior if calling
   -- tasks have different priorities.

  entry Wait_For_N (for Queue in Boolean) when
        not Locked and Hold'Count = 0 and
       (Queue = Current_Queue) and Changed;
  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
   Changed := True;
   Simple_Resource_Controller.Deallocate;
  end Deallocate;

  entry Allocate_N (N : in Natural) when
       Free > 0 and
       not Locked and
       Hold'Count = 0 is
  begin
   if Free >= N then
    Free := Free - N;
    Taken := Taken + N;
   else
    requeue Wait_For_N(Current_Queue);
   end if;
  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
   if Wait_For_N(Queue)'Count = 0 then
    Current_Queue := not Current_Queue;
    Changed := False;
   end if;
   if Free >= N then
    Free := Free - N;
    Taken := Taken + N;
   else
    requeue Wait_For_N(not Queue);
   end if;
  end Wait_For_N;
 end Advanced_Resource_Controller;
end Advanced_Controller;
```

10. CONCLUSIONS

This article 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 article has explored the major issues and, where appropriate, has made concrete proposals. Ada is an extremely expressive language with many orthogonal features. The article has shown that the introduction of extensible protected types does not undermine that orthogonality, and that

the proposal fits in well with limited private types, generics, and normal tagged types.

The work presented here, however, has not been without its difficulties. The major one is associated with overridden entries. It is a fundamental principle of object-oriented programming that a child object can build upon the functionality provided by its parent. The child can call its parent to access that functionality, and therefore extend it. In Ada, calling an entry is a potentially suspending operation, and this is not allowed from within a protected object. Hence, overriding entries gives a conflict between the object-oriented and the protected type models. Furthermore, Ada allows an entry to requeue a call to another entry. When the requeued entry is serviced, control is not returned to the entry which issued the requeue request. Consequently, if a parent entry issues a requeue, control is never returned to the child. This again causes a conflict with the object-oriented programming model, where a child is allowed to undertake postprocessing after a parent call. The article has discussed these conflicts in detail and has proposed a range of potential compromise solutions.

Ada 95 is an important language — the only international standard for object-oriented real-time distributed programming. It is important that it continues to evolve. This article has tried to contribute to the growing debate of how best to fully integrate the protected type model of Ada into the object-oriented model. It is clear that introducing extensible protected types is a large change to Ada and one that is only acceptable at the next major revision of the language. Many of the complications come from the ability to override entries. One possible major simplification of the proposal made here would be not to allow these facilities. Entries would be considered "final" (using Java terminology). Such a simplification might lead to an early transition path between current Ada and a more fully integrated version.

ACKNOWLEDGMENTS

The authors gratefully acknowledge the contributions of Oliver Kiddle and Kristina Lundqvist to the ideas discussed in this paper. We also would like to acknowledge the participants at the 9th International Workshop on Real-Time Ada Issues who gave us some feedback on some of our initial ideas.

REFERENCES

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

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

Brinch-Hansen, P. 1972. Structured multiprogramming. *CACM 15,* 7, 574–578.

Briot, J.-P., Guerraoui, R., and Lohr, K.-P. 1998. Concurrency and distribution in object-oriented programming. *ACM Computing Surveys 30,* 3 (Sept.), 291–329.

Burns, A. and Wellings, A. J. 1998. *Concurrency in Ada*, Second ed. Cambridge University Press.

Caromel, D. 1993. Toward a method of object-oriented concurrent programming. *Communications of the ACM 36,* 9, 90–102.

Corradi, A. and Leonardi, L. 1990. Parellism in object-oriented programming languages. In *IEEE Conference on Computer Languages.* 271–280.

FRØLUND, S. 1992. Inheritance of synchronization constraints in cocurrent object-oriented programming languages. In *Proceedings of ECOOP '92, LNCS*. Vol. 615. Springer, 185–196.

HERLIHY, M. AND WING, J. 1994. Linearizability: A correctness criterion for concurrent objects. *ACM Transactions on Programming Languages and Systems 12,* 3, 463–492.

HOARE, C. 1974. Monitors - an operating system structuring concept. *CACM 17,* 10, 549–557.

JOHNSON, R. 2000. Tagged protected types: Inheritance and polymorphism extensions for synchronization and mutual exclusion in Ada. Ph.D. thesis, Colorado Technical University.

KARAORMAN, M. AND BRUNO, J. 1993. Introducing concurrency to a sequential language. *Communications of the ACM 36,* 9, 103–16.

KIDDLE, O. P. AND WELLINGS, A. J. 1998. Extended protected types. In *Proceedings of ACM SIGAda Annual International Conference (SIGAda 98)*. 229–239.

LEA, D. 1997. *Concurrent Programming in Java*. Addison Wesley.

LISKOV, B. AND WING, J. 1994. A behavioral notion of subtyping. *ACM Transactions on Programming Languages and Systems 16,* 6, 1811–1841.

MAIO, A. D., ATKINSON, C., GOLDSACK, S., MADERNA, F., AND MORETON, T. 1989. DRAGOON: An Ada-based object oriented language for concurrent, real-time distributed systems. In *Ada: The Design Choice, Proceedings Ada-Europe Conference, Madrid*, A. Alvarez, Ed. Cambridge University Press, 39–48.

MATSUOKA, S. AND YONEZAWA, A. 1993. Analysis of inheritance anomaly in object-oriented concurrent programming languages. In *Research Directions in Concurrent Object-Oriented Programming*. MIT Press, 107–150.

MEYER, B. 1993. Systematic concurrent object-oriented programming. *Communications of the ACM 36,* 9, 56–80.

MEYER, B. 1997. *Object-Oriented Software Construction*, Second ed. Prentice Hall.

MICHELL, S. AND LUNDQVIST, K. 1999. Extendable dispatchable task communication mechanisms. In *Proceedings of IRTAW9, Ada Letters, Vol XIX(2)*. 54–59.

MITCHELL, S. E. AND WELLINGS, A. J. 1996. Synchronisation, concurrent object-oriented programming and the inheritance anomaly. *Computer Languages 22,* 1, 15–26.

NEWMAN, R. 1998. The classiC programming language and design of synchronous concurrent object oriented languages. *Journal of Systems Architecture 45,* 5, 387–407.

OAKS, S. AND WONG, H. 1997. *Java Thread*. O'Reilly.

TAFT, T. AND DUFF, R., Eds. 1997. *Ada 95 Reference Manual*. Springer-Verlag. (ISBN 3-540-63144-5).

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.

WIRTH, N. 1988. The programming language Oberon. *Software - Practice and Experience 18,* 7, 671–690.

WYATT, B., KAVI, K., AND HUFNAGEL, S. 1992. Parallelism in object-oriented languages: a survey. *IEEE Software 9,* 6, 56–66.

YOKOTE, Y. AND TORORO, M. 1987. Concurrent programming in concurrentsmalltalk. In *Object-Oriented Concurrent Programming*. MIT Press, 129–158.

YONEZAWA, A., BRIOT, J.-P., AND SHIBAYAMA, E. 1986. Object-oriented concurrent programming in ABCL/1. In *ACM SIGPLAN Notices - Proceedings of OOPSLA 86*. 258–268.

Received May 1999; accepted March 2000