Object-Oriented Programming Enhancements in Ada 200Y

S Tucker Taft
SofCheck, Inc. 11 Cypress Drive, Burlington, MA 01803; USA; Tel:+1 781 750 8068; email: stt@sofcheck.com

Abstract

This article provides an overview of four proposed amendments to the Ada standard for possible inclusion in the revision planned for late 2005 or early 2006. Together, these four amendments can be seen as "finishing" the job of integrating object-oriented programming features into Ada.

Keywords. Ada, Object-Oriented Programming, Amendment

1 Introduction

A new revision of the Ada programming language standard is being prepared, with a scheduled completion date of late 2005 or early 2006. As part of this revision, the Ada Rapporteur Group (ARG), a part of the ISO Working Group 9 (WG9), is developing proposed amendments to the standard. Several of these amendments relate to object-oriented programming (OOP). This paper will describe some of these amendments, and the background and rationale for their development.

When Ada 95 was being designed, there was still a fair amount of controversy whether object-oriented programming features should be included in the language at all, because of their generally dynamic nature, and because of concern about whether some of their perceived negative aspects (difficult to test and verify, "weaker" typing model, etc.) might outweigh their claimed positive aspects.

Over the past decade, object-oriented programming has become the dominant programming paradigm, so much so that it is now simply assumed, and debates have moved on to other language and methodology issues (e.g. aspect-oriented programming, extreme programming, highly scalable programming, etc.). Two major new object-oriented programming languages have appeared on the scene, Java and C#. And most colleges and high schools are now teaching an object-oriented programming language in their introductory programming courses.

Hence, there is no longer any significant debate whether adding object-oriented programming to Ada 95 was a good idea. The question that remains is whether the object-oriented programming features of Ada 95 are as usable, effective, and understandable as they should be.

2 Differences Between Ada 95 and Other OOP Languages

Before attempting to answer this question, it is useful to first identify what makes Ada 95's object-oriented programming features different from those of most other OOP languages, both in a positive and a negative sense. There are several important differences:

a) Ada 95 makes a significant and explicit distinction between class-wide types and specific types. This distinction implicitly exists in essentially all OOP languages, but there is rarely a way to talk about it in the source language itself. Instead, depending on context, a type or class name in such a language might represent a single type in the hierarchy (what Ada 95 calls a "specific" type), or it might represent a type and all types derived directly or indirectly from it (what Ada 95 calls a "derivation class of types").

Only when dealing with class-wide types in Ada 95 is there any possibility of dynamic binding. In most other OOP languages, dynamic binding is the default, and static binding requires additional effort, or is simply not available. This makes it more likely in such languages that dynamic binding will be used in places where static binding would have been preferred, and would haveproduced a faster, more verifiable, and more maintainable system.

In Ada 95, because static binding is the default, there will generally be significantly reduced coupling between a derived type and its parent type, allowing the parent operations to be treated more like black boxes. In most other OOP languages, you really need to see the source code for all parent operations to know whether it is safe to inherit any one of them rather than override it in a derived type.

b) Ada 95 has no direct linguistic support for type hierarchies involving multiple inheritance. Although there are several other language features (such as "with" and "use" clauses, generic packages, private extensions, and access discriminants), that allow programmers to solve problems in Ada 95 for which other languages might rely on their linguistic multiple inheritance capabilities, there are still some situations where

the lack of linguistic support does restrict the ease of solving an important problem.

c) Except for synchronizing operations (such as a task entry call or a protected operation), all operands to an operation in Ada 95 are treated symmetrically in the syntax. That is, they are all passed as parameters "inside" the parentheses, independent of whether the operand might control dynamic binding.

This symmetry makes object-oriented abstract data types be a natural generalization of "normal" abstract data types, and makes user-defined binary operators work in a natural way with such types, without any special treatment. The controlling operand of a binary operator could be the right operand or the left operand, depending on what is appropriate. The controlling "operand" can even be provided by context, in the case of a call on a parameterless function like "Empty_Set" which will result in the invocation of the "appropriate" overriding of Empty_Set, depending on the underlying run-time "tag" of the "receiver" of the result of the call.

Unfortunately, this symmetric approach can result in extra verbiage and possible confusion when used with a multi-package type hierarchy Some "operations" in such a hierarchy might be so-called "class-wide" operations, which are generally declared in the package where the root type of the hierarchy is declared, while others will be "dispatching" operations which are inherited down the hierarchy, and are implicitly declared within each package where a derived type is declared. To call an operation, one has to either have "use" clauses for all packages where it might have been declared, or determine the correct package and put a prefix on the operation name that identifies the relevant package. Although this does not at first glance seem onerous, when working with relatively large type hierarchies, always identifying the package or "use"ing all the relevant packages can make the code less rather than more readable.

With OOP languages that use the "asymmetric" approach, where the (one and only) controlling operand precedes the name of the operation, and the other operands appear inside the parentheses, there is rarely a need to identify the module where an operation is declared, since it is determined by the type of the controlling operand. In C++, the module name is used generally only when overriding the default dynamic binding, and requesting static binding to an operation in a particular class/namespace.

There are some languages, in particular Modula-3, which allow either notation to be used, with the asymmetric "prefix" notation being a short-hand (syntactic "sugaring") for the symmetric notation.

d) Ada separates declaration from implementation, and requires that all types and operations be declared before they are referenced. In some OOP languages, in particular Eiffel and Java, declaration and implementation are not separated in the definition of a class. Furthermore, in these languages, in part because all objects are referenced via pointers and hence are of known "size," there is no need to declare a class before it is referenced.

Because Ada requires declaration before reference, extra work is required to create collections of types that are mutually dependent. In general, an incomplete type declaration is required to allow for such cyclic type structures. However, an incomplete type must be completed within the same package in which it is declared. This precludes such cyclic type structures from crossing multiple packages, and tends to lead to larger-than-ideal packages simply to accommodate such a cycle. The child library unit feature was added to Ada 95 in part to allow packages to remain smaller, with hierarchies (subsystems) of packages being used to represent large multi-type abstractions.

C++ retains the separation between declaration and implementation, while allowing cyclic type structures to cross multiple "namespaces." This is possible because namespaces may be defined in several separate textual pieces, and an incomplete type declaration in C++ may be in one piece of the namespace, while a separate piece contains the full type declaration. In Ada, packages have only two textually separable pieces, namely the package declaration ("spec") and the package implementation ("body"). But putting a full type declaration in the package body is not a solution to the multi-package cyclic type structure problem, because the declarations within the package body are not visible outside the package. By contrast, all the "pieces" of a C++ namespace can contain "visible" declarations.

e) Ada 95 supports 3 levels of visibility for operations and components of a type: fully public, visible to child units, and visible only within the defining package. Most other OOP languages provide special visibility of operations to derived types (subclasses). In C++ and Java this is called "protected" visibility.

An important advantage of the Ada 95 approach to "partial" visibility is that it is provided only to modules whose position within the naming hierarchy implies their special visibility. This creates a strong boundary around the set of units that might be affected by changes to partially

visible operations or components. In most other OOP languages, this special visibility is unrelated to the module structure, and a derived type/subclass which might be affected by changes to partially visible operations or components could be in any module, anywhere in the system.

The net effect is that encapsulation and information hiding in Ada 95 is linked more closely to the naming hierarchy, making maintenance of Ada object-oriented systems easier to perform, even when the systems grow large and involve large hierarchies of types.

f) Ada 95 supports both object-oriented programming and multi-threaded programming, but does not directly integrate these two. Tasks and protected objects can be components of an object-oriented "type," or vice-versa, but neither tasks or protected objects can themselves be directly extended. By contrast, in Java, which is one of the very few other languages that have linguistic support for both object-oriented programming and multi-threading, synchronizing operations can be added in subclasses, and the types used to represent threads can also similarly be extended using the normal inheritance mechanisms.

An important advantage of Ada's tasking model is that all operations of a protected type or a task synchronize properly with one another, while in Java, it is possible to have both synchronizing and non-synchronizing operations on the same type, which is an obvious avenue for subtle race conditions to enter a system. Furthermore, because Ada's protected and task types do not allow piecemeal inheritance, all operations that synchronize with one another are defined in the same module, preserving the original advantages of the "monitor" concept introduced many years ago -- analysis and verification of proper synchronization conditions can be performed without having to chase down all critical sections that might be scattered about the system.

Given the above important differences between Ada 95 and most OOP languages, it is appropriate to evaluate these differences, and see whether they represent strengths or weaknesses in Ada's support for object-oriented programming. In some cases, the differences have both positive and negative aspects. Arguably one overall negative aspect of such differences is that they may put Ada 95 out of the mainstream of object-oriented programming, given that more and more programmers are being introduced to OOP, or even programming as a whole, through languages like Java and C#. On the other hand, Ada 95 has an important role in the development of complex, critical systems, and some of the differences are specifically designed to assist in the development of safe, robust, and verifiable systems, while still providing the flexibility and extensibility of object-oriented programming.

3 Areas of Strength, Areas for Enhancement

The challenge for this upcoming revision of Ada is then to preserve Ada's great strengths in its support for the construction of safe, verifiable systems, while enhancing its object-oriented features to take advantage of what has been learned about object-oriented programming features over the past ten years. Areas that have been identified for possible enhancement are support for multi-package cyclic type structures, support for multiple-inheritance type hierarchies, support for the "asymmetric" notation for invoking operations, and support for some kind of extension for protected and task types.

On the other hand, Ada's clear distinction between specific and class-wide types, its default of static binding with dynamic binding only where necessary, and its strong boundary around modules that have visibility on "partially" visible operations and components, are seen as clear advantages to Ada's approach to object-oriented programming, with no need for significant alteration. Furthermore, any changes that are proposed must not compromise Ada's strengths, and if anything, should extend Ada's unique position as the safe and verifiable, real-time object-oriented programming language.

4 Cyclic Type Structures

One item identified as very important for enhancement has to do with allowing cyclic type structures to cross package boundaries. In Ada 95, it is possible to use a combination of class-wide types, type extension, and "downward" type conversion, to overcome the basic Ada limitation to single-package cyclic type structures. However, this approach introduces additional complexity and some degree of run-time overhead and possible sources of run-time errors. Hence, there has been a concerted effort to provide a natural way for cyclic type structures to be safely and securely extended across packages.

Several alternative proposals have been developed and evaluated. Unfortunately, no one proposal has emerged as clearly the best solution in every dimension. The original proposal introduced a new kind of "with" clause called the "with type" clause. This allowed a package to refer to a type that would eventually be declared in some other package, but without requiring that that other package be compiled first. A version of this proposal was actually implemented in the GNAT Ada Compiler from AdaCore Technologies, but was ultimately dropped from consideration by the ARG because of difficulties discovered while working out the lower level details.

Three proposals remain under consideration: one involving type "stubs" (analogous to program unit "stubs", identified by the "is separate" syntax), a second involving a generalization of incomplete type declarations to allow a parent package to declare an incomplete type that will be

completed in a child or nested package, and a third proposal involving a new kind of "limited" with clause, allowing one package to gain visibility on the types and nested packages of another package, without requiring "full" compilation of the other package.

Here are examples of the three proposals. They all are based on the Employee/Department problem, where there is a type that represents employees, and a type that represents departments, and employees are members of a department, while a department has a manager who is an employee. The challenge is to define the employee type in one package, and the department type in a separate package, but accommodate the desire to have references to both employees and departments in both packages.

The first example is the "type stub" proposal:

```
limited with Employees;  -- Allow type stubs to refer to
                         -- this package
package Departments is
   type Employee is separate Employees.Employee;
        -- Type stub
   type Employee_Ref is access Employee;
   type Department is private;
   procedure Set_Manager(Dept: in out Department;
        Mgr: Employee_Ref);
   function Manager(Dep: Department)
        return Employee_Ref;
   ...
private
   type Department is record
      Mgr: Employee_Ref;
      ...
   end record;
end Departments;

limited with Departments;
        -- Allow type stubs to refer to this package
package Employees is
   type Department is
    separate Departments.Department; -- Type stub
   type Department_Ref is access Department;
   type Employee is private;
   procedure Set_Department(Emp: in out Employee;
        Dept: Department_Ref);
   function Department(Emp: Employee) return
Department_Ref;

   ...
private
   type Employee is record
      Dept: Department_Ref;
      ...
   end record;
end Employees;
```

The second example uses the generalized incomplete type declaration:

```
package Office is
   type Employees.Employee;
                -- Incomplete type completed in child
      type Employee_Ref is
        access Employees.Employee;

   type Departments.Department;
                -- Incomplete type completed in child
      type Department_Ref is
        access Departments.Department;
end Office;

package Office.Departments is
   type Department is private;
   procedure Set_Manager(Dept: in out Department;
        Mgr: Employee_Ref);
   function Manager(Dep: Department)
        return Employee_Ref;
   ...
private
   type Department is record
      Mgr: Employee_Ref;
      ...
   end record;
end Office.Departments;

package Office.Employees is
   type Employee is private;
```

```
      procedure Set_Department(Emp: in out Employee;
          Dept: Department_Ref);
      function Department(Emp: Employee)
          return Department_Ref;

      ...
   private
      type Employee is record
         Dept: Department_Ref;
         ...
      end record;
   end Office.Employees;
```

The third example uses the "limited with" clause:

```
   limited with Employees;
          -- Gives visibility on types as incomplete types
   package Departments is
      type Employee_Ref is access Employees.Employee;
      type Department is private;
      procedure Set_Manager(Dept: in out Department;
          Mgr: Employee_Ref);
      function Manager(Dep: Department)
          return Employee_Ref;
      ...
   private
      type Department is record
         Mgr: Employee_Ref;
         ...
      end record;
   end Departments;


   limited with Departments;
          -- Gives visibility on types as incomplete types
   package Employees is
      type Department_Ref is
          access Departments.Department;

      type Employee is private;

      procedure Set_Department(Emp: in out Employee;
          Dept: Department_Ref);
      function Department(Emp: Employee)
          return Department_Ref;

      ...
   private
      type Employee is record
         Dept: Department_Ref;
         ...
      end record;
   end Employees;
```

All three proposals allow a type defined in one package to be treated as an incomplete type in some other package, without the second package "depending" semantically on the first package. This is the critical capability, because it allows a cyclic type structure to be established without contradicting the partial ordering implied by "normal" semantic dependence relationships. All of the solutions involve a "weaker" kind of dependence, where one package knows that another package "exists" without having full semantic dependence on it. The "limited" with clause proposal approaches this problem by introducing a "limited" dependence on another package. Limited dependences are allowed to be cyclic. They imply some kind of pre-scan of a package to determine the names of the types (and the subpackages) of the package, without doing a full semantic analysis of the package.

The type stub proposal also requires a similar kind of limited dependence, but limits it even further to specific types identified by type stubs. Further, it does not require any kind of pre-scan of the package, because post-compilation checks can be performed to verify that type stubs refer only to types that actually exist in the package.

The incomplete-type-completed-in-a-child proposal introduces a "weak" dependence between a parent package and one of its child packages, requiring that a child package exist and that it declare a type that matches one identified in a generalized form of incomplete type declaration present in the parent's specification.

At this point there is consensus that a solution to this problem will exist in the Ada 200Y standard, and that the form of the solution will be based on one of these three proposals, but the particular approach has not yet been chosen. It is anticipated that the final choice will be made at the ARG meeting immediately following the AdaEurope 2003 conference.

5 Multiple-Inheritance Type Hierarchies

When Ada 95 was designed, a significant amount of energy was expended in evaluating the possibility of including direct syntactic support for multiple inheritance. At the time, some OOP languages included full multiple inheritance (C++, Eiffel), while others chose single

inheritance (Modula-3, Smalltalk). Full multiple inheritance introduces a number of language complexities as well as a somewhat more complicated and/or less efficient run-time model for dispatching calls. Ultimately, we decided to stick with the simplicity of single inheritance for Ada 95, but provide various "building blocks" that could be used to solve problems that in other languages might require multiple inheritance.

Since the Ada 95 design was finalized, a middle ground in the spectrum of inheritance models has become popular that provides multiple inheritance of interfaces (i.e. contracts), but with actual implementation "code" and data components inherited from only a single "primary" parent type. This approach, as exemplified in Java, C#, and to some extent CORBA IDL, eliminates much of the complexity of "full" multiple inheritance, because data components can continue to use the straightforward linear extension approach of single inheritance, and because conflicts due to inheriting code from multiple parent types cannot occur.

The current proposal for adding multiple inheritance of interfaces adds a new kind of type to Ada called an "interface". An interface type is in most respects equivalent to a type declared as "type T is abstract tagged null record;" though the syntax is shortened to be simply "type T is interface;". However, in addition to being usable in all contexts where such an abstract type may be used, the type may also be used as a "secondary" parent type in the declaration of a type extension. Secondary parents ("interface parents") are identified by appearing second or later in a list of the parent types in a record extension. The parent type names are separated from one another by "and", as in:

```
type NT is new Primary and Secondary_1 and
  Secondary_2 and ... with ...;
```

Note that the Primary parent may also be an interface type, since an interface type may be used anywhere an abstract tagged type make be used.

Interfaces may also be used as "parents" of other interfaces, using the following form:

```
type NI is interface with Int_Parent1 and Int_Parent2
  and ...;
```

As implied above, no code or components are inherited from interfaces, only the specification of operations that must be implemented by the type that has the interface as a parent. If an interface has other interfaces as parents, then the union of all the operations of the parents combined with the operations defined on the new interface must be implemented by all (non-abstract) types derived from the new interface.

Here is a larger example which uses interfaces:

```
package MVC is
   -- Set of interfaces that define a
   -- model-view-controller structure.
   type Observer is interface;
      -- "interface" is roughly equivalent to
       -- "abstract tagged null record"
   type Observer_Ref is access all Observer'Class;
     -- An observer waits for changes to a model

   type Model is interface;
   type Model_Ref is access all Model'Access;
     -- A model represents some data structure
     -- that is being viewed and/or manipulated

   procedure Notify(Obs: access Observer;
     M: Model_Ref) is abstract;
     -- Notify observer that model it was observing
     --  has changed
   type View is interface with Observer;
   type View_Ref is access all View'Class;
     -- A view is a visual display of some model
   procedure Display_View(V: access View;
     M: Model_Ref) is abstract;
     -- Display view of associated model

   type Controller is interface with Observer;
   type Controller_Ref is access all Controller'Class;
     -- A controller supports input device(s) for
     --  manipulating/updating an underlying model
   procedure Start_Controller(Ctlr: access Controller;
     M: Model_Ref) is abstract;
     -- Initiate controller for associated model

   procedure Register_View(M: access Model;
     V: View_Ref) is abstract;
     -- Register view for given model.

   procedure Register_Controller(M: access Model;
     Ctlr: Controller_Ref) is abstract;
```

```
   -- Register controller on given model.

end MVC;

with MVC;
with Devices;
package Inputs is
   type Mouse is new Devices.Device with private;

   type Mouse_Controller is new Devices.Device and
    MVC.Controller with private;
    -- Primary parent type, if any, must be listed first
    -- All other parent types must be interfaces.

   procedure Handle_Input(
    MC: in out Mouse_Controller);
    -- Optionally override operations of parent type
     -- (or may inherit those with appropriate defaults)

    procedure Notify(MC: access Mouse_Controller;
      M: Model_Ref);
    procedure Start_Controller(
      MC: access Mouse_Controller; M: Model_Ref);
    -- Required to define all abstract operations
    -- declared for Observer and Controller

   type Two_Button_Mouse_Controller is new
     Mouse_Controller with private;
    procedure Start_Controller(
     TMC: access Two_Button_Mouse_Controller;
      M: Model_Ref);
    -- May inherit or override operations inherited from
    -- parent type including those that are needed for
    -- interfaces Observer and Controller

   procedure Register_And_Start(
    MC: access Mouse_Controller'Class;
    M: Model_Ref);
       -- Class-wide operation to register the mouse
       -- controller on given model, and then start the
            -- controller going.
   private
      ...
   end Inputs;
```

Although not illustrated in the above example, the proposal for interface types includes a proposal for "declared-null" procedures. A declared-null procedure is one whose specification ends with "is null;" rather than ";" or "is abstract;". No separate body is permitted for such a procedure. The implicit null body has no effect when executed.

Rather than requiring that all primitive operations of an interface type be abstract, this proposal also allows the primitive operations to be declared null. Such a procedure need not be overridden in a type derived from this interface. If not overridden, its implementation is null. If at least one interface ancestor of a type declares a given operation as null, the type need not provide an explicit overriding of the operation. If a non-interface ancestor type provides a non-null implementation of the operation, that is inherited rather than the null procedure.

Declared-null procedures are useful in that they allow a number of optional capabilities to be supported in an interface, without every derived type having to explicitly define the capability. In addition, if an abstract or interface type with one or more declared-null primitives is used as the ancestor in a generic formal type extension, the formal type is presumed to have non-abstract implementations of these operations. This can be useful when overriding the operations, since it is often desirable to call the parent's operation from an overriding, particular in the case of initializing or finalizing operations.

6 Using Object.Operation Notation

When doing object-oriented programming in Ada 95, the programmer must identify the package in which an operation is declared, along with the various operands. Because dispatching operations are often implicitly declared, identifying the package where they are declared can sometimes be confusing. In addition to dispatching operations, class-wide operations are important in many object-oriented systems. However in Ada, class-wide operations, unlike their equivalent in many other OOP languages, are not inherited along with the dispatching operations. Instead, they are only declared in the original package where they appear.

This distinction in inheritance between dispatching operations and class-wide operations means that it can be a burden to identify the package where an operation of interest is declared, particularly when the choice between making an operation a dispatching operation versus a class-wide operation might be more of an implementation detail than an essential part of the semantics of the operation from a user's point of view. The distinction is generally

important when deriving from a type, but may be irrelevant when using the type.

Programmers familiar with other OOP languages that use an "object.operation(...)" syntax rather than Ada's "package.operation(object, ...)" syntax find this added burden an entry barrier to using Ada for OOP systems, and tends to make the language feel less object-oriented than it truly is. The alternative of inserting "use" clauses for every possible package where an operation might be declared has other negative ramifications.

Given these considerations, a proposal has been developed to allow the use of an "object.operation(...)" syntax as a syntactic shorthand for "package.operation(object, ...)". Originally it was proposed that this syntactic shorthand be available to all kinds of types, whether or not the type is tagged. However, supporting this for both access types and tagged types adds to the complexity of the proposal in certain ways due to the desire to allow implicit dereference (implicit ".all") of the "object" if it is designated by an access value. Implicit dereference is provided in all other places where "." is allowed in the syntax, and it would be inconsistent not to allow it here. Furthermore, this notation is specifically intended to simplify object-oriented programming where there may be multiple relevant packages. When using non-tagged types, the object.operation syntax would not provide as much benefit.

The basic idea of this restricted proposal is that any dispatching operation, or any class-wide operation declared in a package where the corresponding specific type is declared, is eligible for calling via this shorthand, so long as the first formal parameter is a controlling parameter, or is of the class-wide type. When the object.operation syntax is used, the "operation" is looked up first as a component, and then as though the packages where the type and any of its ancestors are declared had been made use-visible. If the object were of an access-to-tagged type, an interpretation using an implicit dereference would also be considered. If there are possible interpretations of "operation" among these packages, it is checked to see if any of them are subprograms where "object", "object.all", or "object'access" could be passed as the first parameter, and any actual parameters given in parentheses after "operation" correspond to the remaining formals.

Here are some examples of use of this shorthand:

```
--Given the MVC and Inputs packages given above:

   M : MVC.Model_Ref;
   V : MVC.View_Ref;
   C : MVC.Controller_Ref;
   MC : aliased Inputs.Mouse_Controller;
begin
   V.Display_View(M);
      -- equiv to MVC.Display_View(V, M);
   MC.Start_Controller(M);
      -- equiv to Inputs.Start_Controller(MC'Access, M);
   MC.Handle_Input;
      -- equiv to Inputs.Handle_Input(MC);
   MC.Register_And_Start(M);
      -- equiv to Inputs.Register_And_Start
      --    (MC'Access, M);
      -- (this is a call on a class-wide op)
```

7 Inheritance of Interfaces for Protected and Task Types

During the Ada 95 design process, it was recognized that type extension might be useful for protected types (and possibly task types) as well as for record types. However, at the time, both type extension and protected types were somewhat controversial, and expending energy on a combination of these two controversial features was not practical.

Since the design, however, this lack of extension of protected types has been identified as a possible target for future enhancements. In particular, a concrete proposal appeared in the May 2000 issue of ACM Transactions on Programming Languages in Systems (ACM TOPLAS[1]), and this has formed the basis for a language amendment (AI-00250).

However, in ARG discussions, the complexity of this proposal has been of concern, and more recently a simpler suggestion was made that rather than supporting any kind of implementation inheritance, interfaces for tasks and protected types might be defined, and then concrete implementations of these interfaces could be provided. Class-wide types for these interfaces would be defined, and calls on the operations (protected subprograms and entries) defined for these interfaces could be performed given only a class-wide reference to the task or protected object.

An important advantage of eliminating inheritance of any code or data for tasks and protected types is that the "monitor"-like benefits of these constructs are preserved. All of the synchronizing operations are implemented in a single module, simplifying analysis and avoiding any inheritance "anomalies" that have been associated in the literature with combining inheritance with synchronization.

The detailed syntax for protected and task interfaces has not been proposed. Here is one possibility:

```
protected interface Queue is
   -- Interface for a protected queue
   entry Enqueue(Elem : in Element_Type) is abstract;
   entry Dequeue(Elem : out Element_Type)
      is abstract;
```

```
   function Length return Natural is abstract;
end Queue;

type Queue_Ref is access all Queue'Class;

protected type Bounded_Queue(Max: Natural) is
  new Queue with
  -- Implementation of a bounded, protectected queue
   entry Enqueue(Elem : in Element_Type);
    entry Dequeue(Elem : out Element_Type);
   function Length return Natural;
private
   Data: Elem_Array(1..Max);
    In_Index: Positive := 1;
   Out_Index: Positive := 1;
   Num_Elems: Natural := 0;
end My_Queue;

task interface Worker is
  -- Interface for a worker task
     entry Queue_To_Service(Q : Queue_Ref)
      is abstract;
end Server;

type Worker_Ref is access all Worker'Class;

task type Cyclic_Worker is new Worker with
  -- Implementation of a cyclic worker task
     entry Queue_To_Service(Q : Queue_Ref);
end Cyclic_Server;

task Worker_Manager is
  -- Task that manages servers and queues.
     entry Add_Worker_Task(W : Worker_Ref);
     entry Add_Queue_To_Be_Serviced(
      Q : Queue_Ref);
end Worker_Manager;

task body Worker_Manager is
     Worker_Array : array(1..100) of Worker_Ref;
     Queue_Array : array(1..10) of Queue_Ref;
     Num_Workers : Natural := 0;
     Next_Worker : Integer := Worker_Array'First;
     Num_Queues : Natural := 0;
     Next_Queue : Integer := Queue_Array'First;
begin
     loop
        select
              accept Add_Worker_Task(
              W : Worker_Ref) do
                 Num_Workers := Num_Workers + 1;
                 Worker_Array(Num_Workers) :=
                   Worker_Ref(W);
              end Add_Worker_Task;
              -- Assign new task a queue to service
              if Num_Queues > 0 then
                 -- Assign next queue to this worker
                 Worker_Array(Num_Workers).
                  Assign_Queue_To_Service(
                  Queue_Array(Next_Queue));
                        -- Dynamically bound entry call

                 -- Advance to next queue
                 Next_Queue := Next_Queue
                  mod Num_Queues + 1;
              end if;
        or
              accept Add_Queue_To_Be_Serviced(
              Q : Queue_Ref);
                 Num_Queues := Num_Queues + 1;
                 Queue_Array(Num_Queues) :=
                   Queue_Ref(Q);
              end Add_Queue_To_Be_Serviced;

              -- Assign queue to worker if
              -- enough workers
              if Num_Workers >= Num_Queues then
                 -- This queue should be given one
                 -- or more workers
                declare
```

```
            Offset : Natural := Num_Queues-1;
          begin
            while Offset < Num_Workers loop
                -- (re) assign queue to worker
                Worker_Array((Next_Worker
                + Offset - Num_Queues)
                 mod Num_Workers + 1).
                   Assign_Queue_To_Service(
                   Queue_Array(Num_Queues));
                    -- Dynamically bound call
                Offset := Offset + Num_Queues;
            end loop;

            -- Advance to next worker
            Next_Worker := Next_Worker
              mod Num_Workers + 1;
          end;
        end if;
    or
        terminate;
    end select;
  end loop;
end Worker_Manager;

My_Queue : aliased Bounded_Queue(Max => 10);
My_Server : aliased Cyclic_Server;

begin
  Worker_Manager.Add_Worker_Task(
    My_Server'Access);
  Worker_Manager.Add_Queue_To_Be_Serviced(
    My_Queue'Access);
```

8 Summary

The four proposed amendments to the Ada standard discussed above are in some sense an attempt to "finish" the job of integrating object-oriented programming into Ada which was started during the Ada 95 revision process. Although the existing OOP features in Ada 95 are both powerful and flexible, eight years of use and ongoing developments in the object-oriented programming language community have suggested opportunities for enhancement.

Although it is likely that some of these amendments will be approved for addition to the standard, it is quite possible that some will not, or that the proposals will be further refined in minor or major ways. Hence it is essential to keep in mind that this is a snapshot of an ongoing revision process, and by no means the final story. For those interested in tracking the progress of these amendments, the website of the Ada Conformance Assessment Authority (ACAA) provides ready access to all of the amendments, as well as minutes of ARG meetings. The URL for this website is:

http://www.ada-auth.org/

References

[1] Wellings, A.J.; Johnson, B.; Sanden, B.; Kienzle, J., Wolf, Th., and Michell, S.: "Integrating Object-Oriented Programming and Protected Objects in Ada 95", ACM TOPLAS 22 (3), May 2000; pp. 506 - 539.