An Invitation to Ada 2005

Pascal Leroy

Rational Software, 1 place Charles de Gaulle, 78067 St Quentin en Yvelines, France
pascal.leroy@fr.ibm.com

Abstract. Starting in 2000, the ISO technical group in charge of maintaining the Ada language has been looking into possible changes for the next revision of the standard, around 2005. Based on the input from the Ada community, it was felt that the revision was a great opportunity for further enhancing Ada by integrating new programming practices, e.g., in the OOP area; by providing new capabilities for embedded and high-reliability applications; and by remedying annoyances encountered during many years of usage of Ada 95. This led to the decision to make a substantive revision rather than a minor one. This paper outlines the standardization process and schedule, and presents a number of key improvements that are currently under consideration for inclusion in Ada 2005.

1 Introduction

Every ISO standard is reviewed every five years to determine if it should be confirmed (kept as is), revised, or withdrawn. The international standard for Ada [1], known as ISO/IEC 8652 in ISO jargon, was published in 1995. A Technical Corrigendum, ISO/IEC 8652 Corr. 1, was published at the end of the first review period, in 2001. This document corrected a variety of minor errors or oversights in the original language definition, with the purpose of improving the safety and portability of programs. However, it did not add significant new capabilities.

With 7 years' experience with Ada 95, the ISO working group in charge of maintaining the language (known as JTC1/SC22/WG9) has come to the conclusion that more extensive changes were needed at the end of the second review period, in 2005. As a result, it was decided to develop an Amendment to integrate into the Ada language the result of more that 10 years of research and practice in programming languages. (In ISO terms, an Amendment is a much larger change than a Technical Corrigendum.)

WG9 has asked its technical committee, the Ada Rapporteur Group (ARG) to prepare proposals for additions of new capabilities to the language. Such capabilities could take the form of new core language features (including new syntax), new predefined units, new specialized needs annexes, or secondary standards, as appropriate. The changes may range from relatively minor to quite substantial.

2 Revision Guidelines

As part of the instructions it gave to the ARG, WG9 has indicated that the main purpose of the Amendment is to address identified problems in Ada that are interfering with the usage or adoption of the language, especially in those application areas where it has traditionally had a strong presence: high-reliability, long-lived real-time and embedded applications, and very large complex systems.

In particular, the ARG was requested to pay particular attention to two categories of improvements:

- Improvements that will maintain or improve Ada's advantages, especially in those user domains where safety and criticality are prime concerns;
- Improvements that will remedy shortcomings of Ada.

Improvements that fall into the first category include: new real-time features, such as the Ravenscar profile, and features that improve static error detection. Improvements that fall into the second category include a solution to the problem of mutually dependent types across packages, and support of Java-like interfaces.

In selecting features for inclusion in the Amendment, it is very important to avoid gratuitous changes that could impose unnecessary disruption on the existing Ada community. Therefore, the ARG was asked to consider the following factors when evaluating proposals:

- Implementability: can the proposed feature be implemented at reasonable cost?
- Need: does the proposed feature fulfill an actual user need?
- Language stability: would the proposed feature appear disturbing to current users?
- Competition and popularity: does the proposed feature help improve the perception of Ada, and make it more competitive with other languages?
- Interoperability: does the proposed feature ease problems of interfacing with other languages and systems?
- Language consistency: is the provision of the feature syntactically and semantically consistent with the language's current structure and design philosophy?

In order to produce a technically superior result, it was also deemed acceptable to compromise strict backward compatibility when the impact on users is judged to be acceptable. It must be stressed, though, that the ARG has so far been extremely cautious on the topic of incompatibilities: incompatibilities that can be detected statically (e.g., because they cause compilation errors) might be acceptable if they are necessary to introduce a new feature that has numerous benefits; on the other hand, incompatibilities that would silently change the effect of a program are out of the question at this late stage.

In organizing its work, the ARG creates study documents named Ada Issues (AI) that cover all the detailed implications of a change. The Ada Issues may be consulted on-line at http://www.ada-auth.org/ais.html. The reader should keep in mind though that these are working documents that are constantly evolving, and that there is no guarantee that any particular AI will be included in the final Amendment.

3 Revision Schedule

At the time of this writing, WG9 targets the following schedule for the development of the Amendment:

- September 2003: receipt of the final proposals from groups other than WG9 or delegated bodies.
- December 2003: receipt of the final proposals from WG9 or delegated bodies.
- June 2004: approval of the scope of the Amendment, perhaps by approving individual AIs, perhaps by approving the entire Amendment document.
- Late 2004: informal circulation of the draft Amendment document, receipt of comments, and preparation of final text.
- Spring 2005: completion of proposed text of the Amendment.
- Mid 2005: WG9 email ballot.
- 3Q2005: SC22 ballot (SC22 is the parent body of WG9, and is in charge of the standardization of all programming languages).
- Late 2005: JTC1 ballot, final approval.

The elaboration of the Amendment being a software-related project, and a very complex one to boot, it would not be too surprising if the above schedule slipped by a couple of months. However, the work will essentially be schedule-driven, and the ARG will effectively stop developing new features at the beginning of 2004. The following two years will be spent writing and refining the Amendment document to ensure that its quality is on a par with that of the original Reference Manual.

4 Proposed New Features

The rest of this paper gives a technical overview of the most important AIs that are currently being considered for inclusion in the Amendment. Again, this is just a snapshot of work currently in progress, and there is no guarantee that all of these changes will be included in the Amendment. And even if they are, they might change significantly before their final incarnation.

For the convenience of the reader, the following table lists the proposals that are discussed below, in the order in which they appear:

General-Purpose Capabilities
Handling Mutually Dependent Type across Packages
Access to Private Units in the Private Part
Downward Closures for Access to Subprograms
Aggregates for Limited Types
Pragma Unsuppress
Real-Time, Safety and Criticality
Ravenscar Profile for High Integrity Systems
Execution-Time Clocks

Object-Oriented Programming

Abstract Interface to Provide Multiple Inheritance
Generalized Use of Anonymous Access Types
Accidental Overloading When Overriding

Programming By Contract

Pragma Assert, Pre-Conditions and Post-Conditions

Interfacing with Other Languages or Computing Environments

Unchecked Unions: Variant Records with No Run-Time Discriminant
Directory Operations

4.1 Handling Mutually Dependent Types across Packages (AI 217)

The impossibility of declaring mutually dependent types across package boundaries has been identified very early after the standardization of Ada 95. This problem, which existed all along since Ada 83, suddenly became much more prominent because the introduction of child units and of tagged types made it more natural to try and build interdependent sets of related types declared in distinct library packages.

A simple example will illustrate the problem. Consider two classes, Department and Employee, declared in two distinct library packages. These classes are coupled in two ways. First, there are operations in each class that take parameters of the other class. Second, the data structures used to implement each class contain references to the other class. The following (incorrect) code shows how we could naively try to represent this in Ada:

```
with Employees;
package Departments is
   type Department is tagged private;
   procedure Choose_Manager
                (D       : in out Department;
                 Manager : in out Employees.Employee);
private
   type Emp_Ptr is access all Employees.Employee;
   type Department is tagged
      record
         Manager : Emp_Ptr;
         …
      end record;
end Departments;
```

```
with Departments;
package Employees is
   type Employee is tagged private;
   type Dept_Ptr is access all Departments.Department;
   procedure Assign_Employee
                (E : in out Employee;
                 D : in out Departments.Department);
   function Current_Department (D : in Employee)
                                return Dept_Ptr;
```

```
private
   type Employee is tagged
      record
         Department : Dept_Ptr;
         …
      end record;
end Employees;
```

This example is patently illegal, as each specification has a with clause for the other one, so there is no valid compilation order.

The ARG has spent considerable effort on this issue, which is obviously a high-priority one for the Amendment. It is unfortunately very complex to solve, and at the time of this writing three proposals are being considered: type stubs, incomplete types completed in children, and limited with clauses. These proposals will be quickly presented in the sections below (in no particular order).

All proposals have in common that they somehow extend incomplete types to cross compilation units: incomplete types are the natural starting point for solving the problem of mutually dependent types, since they are used for that purpose (albeit in a single declarative part) in the existing language. When incomplete types are used across compilation units' boundaries, the normal limitations apply: in essence, it is possible to declare an access type designating an incomplete type, and not much else.

In addition, all proposals introduce the notion of tagged incomplete type. For a tagged incomplete type T, two additional capabilities are provided. First, it is possible to reference the class-wide type T'Class. Second, it is possible to declare subprograms with parameters of type T or T'Class. The reason why tagged incomplete types may be used as parameters is that they are always passed by-reference, so the compiler doesn't need to know their physical representation to generate parameter-passing code: it just needs to pass an address.

4.1.1 Type Stubs

A type stub declares an incomplete type that will be completed in another unit, and gives the name of the unit that will contain the completion:

```
type T1 is separate in P;
type T2 is tagged separate in P;
```

These declarations define two incomplete types, T1 (untagged) and T2 (tagged) and indicate that they will be completed in the library package P.

The rule that makes it possible to break circular dependencies is that, when the above type stubs are compiled, the package P doesn't have to be compiled. Later, when P is compiled, the compiler will check that it declares an untagged type named T1 and tagged type named T2.

For safety reasons, a unit that declares type stubs like the above must include a new kind of context clause to indicate that it contains a forward reference to package P. The syntax for doing this is:

```
abstract of P;
```

The package that provides the completion of the type stub, P, must have a with clause for the unit containing the stub. Moreover, each type may have at most one type stub (this is checked at post-compilation time).

With type stubs, the example presented above can be written as follows:

```
abstract of Departments;
package Departments_Interface is
   type Department is tagged separate in Departments;
end Departments_Interface;

abstract of Employees;
package Employees_Interface is
   type Employee is tagged separate in Employees;
end Employees_Interface;

with Departments_Interface, Employees_Interface;
package Departments is
   type Department is tagged private;
   procedure Choose_Manager
               (D       : in out Department;
                Manager : in out
                   Employees_Interface.Employee);
private
   type Emp_Ptr is access all
                  Employees_Interface.Employee;
   … -- As above.
end Departments;

with Departments_Interface, Employees_Interface;
package Employees is
   type Employee is tagged private;
   type Dept_Ptr is access all
                   Departments_Interface.Department;
   procedure Assign_Employee
               (E : in out Employee;
                D : in out
                    Departments_Interface.Department);
   function Current_Department (D : in Employee)
                               return Dept_Ptr;
private
   … -- As above.
end Employees;
```

Here we first declare two auxiliary packages, Departments_Interface and Employees_Interface, which contain the stubs. The context clauses make it clear that each package acts as the "abstract" of Departments and Employees, respectively. These packages are compiled first, and they declare two incomplete types, which will be completed in Departments and Employees.

Then Departments and Employees may be compiled, in any order, as they don't depend semantically on each other (although chances are that each body will depend semantically on the other package). Each package has a with clause for both Departments_Interface and Employees_Interface, and this requires an explanation. Take the case of Departments. Obviously it needs a with clause for Employees_Interface in order to declare the second parameter of Choose_Manager and the access type Emp_Ptr,

using the tagged incomplete type Employees_Interface.Employee. But it also needs a with clause for Departments_Interface, as it completes a type stub declared there.

Note that in order to break the circularity it would be sufficient to declare only one type stub, say for type Department, for use by package Employees. Then package Departments could have a direct with clause for Employees. Here we have chosen to keep the example symmetrical, and to create two type stubs.

4.1.2 Incomplete Types Completed in Child Units

Another approach that is being considered is to allow incomplete types to be completed in child units. This can be thought of as a restricted version of type stubs, as the stub and the completion may not be in unrelated packages. The rationale for imposing this restriction is that types that are part of a cycle are tightly related, so it makes sense to require that they be part of the same subsystem. This somewhat simplifies the user model, as it doesn't require so much new syntax to declare the stub and tie it to its completion. Of course, this simplification comes at the expense of some flexibility, as all types that are part of a cycle have to be under the same root unit.

The syntax used to declare such an incomplete type is simply:

```
type C.T1;
type C.T2 is tagged;
```

These declarations define two incomplete types, T1 (untagged) and T2 (tagged) and indicate that they will be completed in unit C, a child of the current unit. As usual, the normal restrictions regarding the usages of incomplete types before their completion apply.

With incomplete types completed in child units, our example can be written as follows:

```
package Root is
   type Departments.Department;
   type Employees.Employee;
end Root;

package Root.Departments is
   type Department is tagged private;
   procedure Choose_Manager
                (D       : in out Department;
                 Manager : in out Employees.Employee);
private
   type Emp_Ptr is access all Employees.Employee;
   … -- As above.
end Root.Departments;

package Root.Employees is
   type Employee is tagged private;
   type Dept_Ptr is access all Departments.Department;
   procedure Assign_Employee
                (E : in out Employee;
                 D : in out Departments.Department);
   function Current_Department (D : in Employee)
                               return Dept_Ptr;
```

```
private
   … -- As above.
end Root.Employees;
```

Here we first declare a root unit, Root, to hold the two incomplete type declarations. Then we can write the two child units in a way which is quite close to our original (illegal) example. Note however that in package Root.Departments the name Employees.Employee references the incomplete type declared in the parent, not the full type declared in the sibling package Root.Employees. Therefore, the two children don't depend semantically on each other, and the circularity is broken. However, the usages of Employees.Employee in Root.Departments are restricted to those that are legal for a (tagged) incomplete type.

4.1.3 Limited With Clauses

A different approach is to look again at the original (illegal) example, and see what it would take to make it legal. Obviously we cannot allow arbitrary circularities in with clauses, but when we look at the specifications of Departments and Employees, we see that they make a very restricted usage of the types Employees.Employee and Departments.Department, respectively: they only try to declare parameters and access types. If we introduce a new kind of with clause that strictly restricts the entities that are visible to clients and how they can be used, we can allow circularities among these new with clauses.

A limited with clause has the following syntax:

```
limited with P, Q.R;
```

A "limited with" clause gives visibility on a *limited* view of a unit. A limited view is one that contains only types and packages (excluding private parts), and where all types appear incomplete. Thus, a limited view of the original Departments package looks as follows:

```
package Departments is
   type Department is tagged;
end Departments;
```

The rule that makes it possible to break circularity is that a limited with clause doesn't introduce a semantic dependence on the units it mentions. Thus limited with clauses are allowed in situations where normal with clauses would be illegal because they would form cycles.

Using limited with clauses, our example can be written as follows:

```
limited with Employees;
package Departments is
   type Department is tagged private;
   procedure Choose_Manager
                (D       : in out Department;
                 Manager : in out Employees.Employee);
private
   type Emp_Ptr is access all Employees.Employee;
   … -- As above.
end Departments;
```

```
limited with Departments;
package Employees is
   type Employee is tagged private;
   type Dept_Ptr is access all Departments.Department;
   procedure Assign_Employee
                (E : in out Employee;
                 D : in out Departments.Department);
   function Current_Department (D : in Employee)
                                return Dept_Ptr;
private
   … -- As above.
end Employees;
```

Here we have simply changed the normal with clauses to be limited with clauses. In package Employees, the limited with clause for Departments gives visibility on the limited view shown above, so the type Department may be named, but it is a tagged incomplete type, and its usages are restricted accordingly. The situation is symmetrical in package Departments.

Internally, the compiler first processes each unit to create the limited views, which contain incomplete type declarations for Department and Employee. During this first phase the with clauses (limited or not) are ignored, so any ordering is acceptable. Then the compiler analyzes each unit a second time and does the normal compilation processing, using the normal compilation ordering rules. During this second phase the limited with clauses give visibility on the limited views, and therefore on the incomplete types. This two-phase processing is what makes it possible to find a proper compilation ordering even in the presence of cycles among the limited with clauses.

4.2 Access to Private Units in the Private Part (AI 262)

The private part of a package includes part of the implementation of the package. For instance, the components of a private type may include handles and other low-level data structures.

Ada 95 provides private packages to organize the implementation of a subsystem. Unfortunately, these packages cannot be referenced in the private part of a public package—the context clause for the private package is illegal. This makes it difficult to use private packages to organize the implementation details of a subsystem.

AI 262 defines a new kind of with clause, the "private with" clause, the syntax of which is:

```
private with A, B.C, D;
```

This clause gives visibility on the entities declared in units A, B.C, and D, but only in the private part of the unit where the clause appears. Thus, a public unit may reference a private unit in a private with.

The following is an example derived from code in the CLAW library (which provides a high-level interface over the Microsoft® Windows® UI). The low-level interface for an image list package looks like:

```
private package Claw.Low_Level_Image_Lists is
   type HImage_List is new DWord;
   type IL_Flags is new UInt;
   ILR_DEFAULTCOLOR : constant IL_Flags := 16#0000#;
   ILR_MONOCHROME   : constant IL_Flags := 16#0001#;
   …
end Claw.Low_Level_Image_Lists;
```

The high-level interface for the image list package looks like:

```
private with Claw.Low_Level_Image_Lists;
package Claw.Image_List is
   type Image_List_Type is tagged private;
   procedure Load_Image
               (Image_List : in out Image_List_Type;
                Image      : in String;
                Monochrome : in Boolean := False);
   …
private
   type Image_List_Type is tagged
     record
       Handle : Claw.Low_Level_Image_Lists.HImage_List;
       Flags : Claw.Low_Level_Image_Lists.IL_Flags;
       …
     end record;
end Claw.Image_List;
```

Here the private part of the high-level package needs to reference declarations from the low-level package. Because the latter is a private child, this is only possible with a private with clause.

4.3 Downward Closures for Access to Subprogram Types (AI 254)

One very important improvement in Ada 95 was the introduction of access-to-subprogram types, which make it possible to parameterize an operation by a subprogram. However, this feature is somewhat limited by the accessibility rules: because Ada (unlike, say, C), has nested subprograms, the language rules must prevent the creation of dangling references, i.e., access-to-subprogram values that outlive the subprogram they designate.

Consider for example the case of a package doing numerical integration:

```
package Integrate is
   type Integrand is access function (X : Float)
                                     return Float;
   function Do_It (Fn : Integrand; Lo, Hi : Float)
                   return Float;
end Integrate;
```

This package works fine if the function to be integrated is declared at library level. For instance, to integrate the predefined square root function, one would write:

```
Result := Integrate.Do_It
            (Ada.Numerics.
                Elementary_Functions.Sqrt'Access,
             0.0, 1.0);
```

However, the package Integrate cannot be used with functions that are not declared at library level. One good practical example of this situation is double integration, where the function to be integrated is itself the result of an integration:

```
function G (X : Float) return Float is
   function F (Y : Float) return Float is
   begin
      … -- Returns some function of X and Y.
   end F;
begin
   return Integrate.Do_It (F'Access, -- Illegal.
                           0.0, 1.0);
end G;

Result : Float := Integrate.Do_It (G'Access, 0.0, 1.0);
```

The accessibility rules are unnecessarily pessimistic here, because the integration algorithm does not need to store values of the access-to-subprogram type Integrand, so there is no risk of creating dangling references.

AI 254 proposes to introduce anonymous access-to-subprogram types. Such types can only be used as subprogram parameter types. Because they cannot be used to declare components of data structures, they cannot be stored and therefore cannot be used to create dangling references. With this feature, the package Integrate would be rewritten as:

```
package Integrate is
   function Do_It
               (Fn : access function (X : Float)
                                      return Float;
                Lo, Hi : Float)
                return Float;
end Integrate;
```

Now, the double integration example above would be legal, as there are no accessibility issues with anonymous access types. Note that the syntax may look a bit heavyweight, but it actually follows the one that was used in Pascal.

4.4 Aggregates for Limited Types (AI 287)

Limited types allow programmers to express the idea that copying values of a type does not make sense. This is a very useful capability; after all, the whole point of a compile-time type system is to allow programmers to formally express which operations do and do not make sense for each type.

Unfortunately, Ada places certain limitations on limited types that have nothing to do with the prevention of copying. The primary example is aggregates: the programmer is forced to choose between the benefits of aggregates (full coverage checking)

and the benefits of limited types. These two features ought to be orthogonal, allowing programmers to get the benefits of both.

AI 287 proposes to allow aggregates to be of a limited type, and to allow such aggregates as the explicit initial value of objects (created by object declarations or by allocators). This change doesn't compromise the safety of limited types; in particular, assignment is still illegal, and the initialization expression for a limited object cannot be the name of another object; also, there is no such thing as an aggregate for a task or protected type, or for a limited private type.

Because a limited type may have components for which it is not possible to write a value (e.g., components of a task or protected type), AI 287 also allows a box "<>" in place of an expression in an aggregate. The box is an explicit request to use default initialization for that component.

The following is an example of an abstraction where copying makes no sense, so the type Object should be limited. Actually, this type has a component of a protected type, so it *has* to be limited.

```
package Dictionary is
   type Object is limited private;
   type Ptr is access Object;
   function New_Dictionary return Ptr;
   …
private
   protected type Semaphore is …;
   type Tree_Node;
   type Access_Tree_Node is access Tree_Node;
   type Object is limited
      record
         Sem: Semaphore;
         Size : Natural;
         Root : Access_Tree_Node;
      end record;
end Dictionary;

package body Dictionary is
   function New_Dictionary return Ptr is
   begin
      return new T'(Sem => <>,
                    Size => 0,
                    Root => null);
   end New_Dictionary;
   …
end Dictionary;
```

With the introduction of aggregates for limited types, the allocator in the body of New_Dictionary is legal. Note that in Ada 95, one would have to first allocate the object, and then fill the components Size and Root. This is error-prone, as during maintenance components might be added and not initialized. With limited aggregates, the coverage rules ensure that if components are added, the compiler will flag the aggregates where new component associations must be provided.

4.5 Pragma Unsuppress (AI 224)

It is common in practice for some parts of an Ada program to depend on the presence of the canonical run-time checks defined by the language, while other parts need to suppress these checks for efficiency reasons. For example, consider a saturation arithmetic package. The multiply operation might look like:

```
function "*" (Left, Right : Saturation_Type)
              return Saturation_Type is
begin
   return Integer (Left) * Integer (Right);
exception
   when Constraint_Error =>
      if (Left > 0 and Right > 0) or
         (Left < 0 and Right < 0) then
         return Saturation_Type'Last;
      else
         return Saturation_Type'First;
      end if;
end "*";
```

This function will not work correctly without overflow checking. Ada 95 does not provide a way to indicate this to the compiler or to the programmer.

AI 224 introduces the configuration pragma Unsuppress, which has the same syntax as Suppress, except that it cannot be applied to a specific entity. For instance, to ensure correctness of the above function, one should add, in its declarative part, the pragma:

```
pragma Unsuppress (Overflow_Check);
```

The effect of this pragma is to revoke any suppression permission that may have been provided by a preceding or enclosing pragma Suppress. In other words, it ensures that the named check will be in effect regardless of what pragmas Suppress are added at outer levels.

4.6 Ravenscar Profile for High-Integrity Systems (AIs 249 and 305)

The Ravenscar profile [2] is a subset of Ada (also known as a tasking profile) which was initially defined by the 8th International Real-Time Ada Workshop (IRTAW), and later refined by the 9th and 10th IRTAW. This profile is intended to be used in high-integrity systems, and makes it possible to use a run-time kernel of reduced size and complexity, which can be implemented reliably. It also ensures that the scheduling properties of programs may be analyzed formally.

The Ravenscar profile has been implemented by a number of vendors and has become a *de facto* standard. The ARG plans to include it in the Amendment. However, in looking at the details of the restrictions imposed by Ravenscar, it was quickly realized that many of them would be generally useful to Ada users, quite independently of real-time or high-integrity issues. So the ARG decided to proceed in two phases: first, it added to the language a number of restrictions and a new configuration pragma;

then it defined the Ravenscar profile uniquely in terms of language-defined restrictions and pragmas. This approach has three advantages:

- Current users of the Ravenscar profile are virtually unaffected.
- Users who need to restrict the usage of some Ada constructs without abiding by the full set of restrictions imposed by Ravenscar can take advantage of the newly defined restrictions.
- Implementers may define additional profiles that cater to the needs of particular users communities.

AI 305 defines new restrictions corresponding to the Ada constructs that are outlawed by the Ravenscar profile. To take only two examples, restrictions No_Calendar and No_Delay_Relative correspond to the fact that Ravenscar forbids usage of package Ada.Calendar and of relative delay statements. AI 305 also defines a new configuration pragma, Detect_Blocking, which indicates that the runtime system is required to detect the execution of potentially blocking operations in protected operations, and to raise Program_Error when this situation arises.

AI 249 then defines a new configuration pragma, Profile, which specifies as its argument either the predefined identifier Ravenscar or some implementation-defined identifier (this is to allow compiler vendors to support additional profiles). The Ravenscar profile is defined to be equivalent to a list of 19 restrictions, to the selection of policies Ceiling_Locking and FIFO_Within_Priorities, and to the detection of blocking operations effected by pragma Detect_Blocking.

4.7 Execution-Time Clocks (AI 307)

The 11th IRTAW identified as one of the most pressing needs of the real-time community (second only to the standardization of the Ravenscar profile) the addition of a capability to measure execution time.

AI 307 proposes to introduce a new predefined package, Ada.Real_Time.Execution_Time, exporting a private type named CPU_Time, which represents the processor time consumed by a task. Arithmetic and conversion operations similar to those in Ada.Real_Time are provided for this type. In addition, it is possible to query the CPU time consumed by a task by calling the function CPU_Clock, passing the Task_ID for that task. This package also exports a protected type, Timer, which may be used to wait until some task has consumed a predetermined amount of CPU time.

Ada.Real_Time.Execution_Time makes it possible to implement CPU time budgeting, as shown by the following example, where a periodic task limits its own execution time to some predefined amount called WCET (worst-case execution time). If an execution time overrun is detected, the task aborts the remainder of its execution, until the next period:

```
with Ada.Task_Identification;
with Ada.Real_Time.Execution_Time;
   …
   use Ada.Real_Time;
   use type Ada.Real_Time.Time;
```

```
…
task body Periodic_Stopped is
   The_Timer : Execution_Time.Timer;
   Next_Start : Real_Time.Time := Real_Time.Clock;
   WCET : constant Duration := 1.0E-3;
   Period : constant Duration := 1.0E-2;
begin
   loop
      The_Timer.Arm
         (To_Time_Span (WCET),
          Ada.Task_Identification.Current_Task);
      select
         -- Execution-time overrun detection.
         The_Timer.Time_Expired;
         Handle_The_Error;
      then abort
         Do_Useful_Work;
      end select;
      The_Timer.Disarm;
      Next_Start := Next_Start +
                    Real_Time.To_Time_Span (Period);
      delay until Next_Start;
   end loop;
end Periodic_Stopped;
```

Here the call to Arm arms the timer so that it expires when the current task has consumed WCET units of CPU time. The loop then starts doing the useful work it's supposed to do. If that work is not completed before the timer expires, the entry call Time_Expired is accepted, the call to Do_Useful_Work is aborted, and the procedure Handle_The_Error is called.

4.8 Abstract Interfaces to Provide Multiple Inheritance (AI 251)

At the time of the design of Ada 95, it was decided that multiple inheritance was a programming paradigm which imposes too heavy a distributed overhead to introduce in Ada, where performance concerns are prevalent.

Since then, an interesting form of multiple inheritance has become commonplace, pioneered notably by Java and COM. An "interface" defines what methods a class must implement, without supplying the implementation of these methods. A class may "implement" any number of interfaces, in which case it must provide implementations for all the methods inherited from these interfaces. Interfaces have the attractive characteristic that they are relatively inexpensive at run-time. In particular, unlike full-fledged multiple inheritance, they do not impose a distributed overhead on programs which do not use them.

The ARG is studying the possibility of introducing interfaces in Ada. This is a sizeable language change, which affects many areas of the language, so this proposal is somewhat less mature than most of the others discussed in this paper.

AI 251 proposes to add new syntax to declare interfaces:

```
type I1 is interface; -- A root interface.
type I2 is interface; -- Another root interface.
```

```
-- An interface obtained by composing I1 and I2:
type I3 is interface with I1 and I2;
```

An interface may have primitive subprograms, but no components. In many respects, it behaves like an abstract tagged type; in particular, constructs which would create objects of an interface type are illegal.

It is expected that most componentless abstract types in Ada 95 could be turned into interfaces with relatively little effort, and with the added flexibility of using them in complex inheritance lattices.

Note that, in order to preserve compatibility, the word "interface" is not a new keyword. It is a new kind of lexical element, dubbed "non-reserved keyword", which serves as a keyword in the above syntax, but is a normal identifier everywhere else.

A tagged type may implement one or more interfaces by using a new syntax for type derivation:

```
type I4 is interface …;
type T1 is tagged …;
type T2 is new T1 and I3 and I4 with
   record
      …
   end record;
```

In this instance, T2 is normally derived from T1 (and it inherits the primitive operations of T1) and implements interfaces I3 and I4. It must therefore override all the primitive operations that it inherits from I3 and I4.

Interfaces become really handy in conjunction with class-wide types. The Class attribute is available for interfaces. Thus, a parameter of type I'Class can at execution be of any specific tagged type that implements the interface I. Similarly, it is possible to declare an access type designating I'Class, and to build data structures that gather objects of various types which happen to implement I.

Because many reusable components in Ada are written as generics, and for consistency with the rest of the language, new kinds of generic formal types are added. They are useful to pass interface types to a generic or to indicate that a formal derived type implements some number of interfaces. Their syntax is similar to that of normal interface and derived type declarations.

In implementation terms, the only operations that may incur a significant performance penalty are membership tests and conversions from a class-wide interface. Say that X is a parameter of type I'Class (I being an interface) and T is a specific tagged type. Evaluating T (X) (a conversion) requires a check at execution time that X is actually in the derivation tree rooted at T. This check already exists for normal tagged types, but it is more complex in the presence of interfaces, because the derivation structure is an acyclic graph, not a tree. Also, as part of the conversion the dispatch table must be changed so as to ensure that dispatching executes the operations of T.

The following example shows a concrete case of usage of interfaces:

```
type Stack is interface;
procedure Append (S : in out Stack;
                  E : Element) is abstract;
procedure Remove_Last (S : in out Stack;
                       E : out Element) is abstract;
function Length (S : Stack) return Natural is abstract;
```

```
type Queue is interface;
procedure Append (Q : in out Queue;
                  E : Element) is abstract;
procedure Remove_First (Q : in out Queue;
                        E : out Element) is abstract;
function Length (Q : Queue) return Natural is abstract;

type Deque is interface with Queue and Stack;
```

Here Stack and Queue are two interfaces which declare the operations (or the contract) characteristic of stacks and queues. Deque is an interface obtained by mixing these two interfaces. It has the characteristic that elements can be removed at either end of the deque.

```
package … is
   type My_Deque is new Deque with private;
   procedure Append (D : in out My_Deque;
                     E : Element;
   procedure Remove_First (D : in out My_Queue;
                           E : out Element);
   procedure Remove_Last (D : in out My_Deque;
                          E : out Element);
   function Length (D : My_Deque) return Natural;
private
   type My_Deque is new Blob and Deque with
      record
         … -- Implementation details here.
      end record;
end …;

procedure Print (S : Stack'Class) is
   E : Element;
begin
   if Length (S) /= 0 then
      Remove_Last (S, E);
      Print (E);
      Print (S);
      Append (S, E);
   end if;
end Print;
```

Here My_Deque is a concrete implementation of Deque, which declares record components as needed, and overrides the subprograms Append, Remove_First, Remove_Last and Length.

The procedure Prints recursively prints a stack by removing its first element, printing it, printing the rest of the stack, and re-appending the first element. It can be used with any type that implements the interface Stack, including Deque and My_Deque (but not with type Queue, which does not have a Remove_Last primitive operation).

4.9 Generalized Use of Anonymous Access Types (AI 230)

In most object-oriented languages, types that are references to a subclass are freely convertible to types that are references to its superclass. This rule, which is always

safe, significantly reduces the need for explicit conversions in code that deals with complex inheritance hierarchies.

In Ada however, explicit conversions are generally required, even when going from a subclass to its superclass. This obscures the explicit conversions that really do need attention (because they may fail a run-time check), and makes the whole object-oriented coding style more cumbersome than in other languages.

Another problem that is somewhat related is that of "named access type proliferation". This commonly occurs when, for one reason or another, an access type is not defined at the point of the type declaration (for instance, a pure package can not declare an access type). Ultimately, if they need to create references, clients end up declaring their own "personal" access type, causing yet more need for unnecessary explicit conversions.

To address these problems, AI 230 proposes to generalize the use of anonymous access types to components and object renaming declarations. As an example, consider the following (toy) object hierarchy:

```
type Animal is tagged …;
type Horse is new Animal with …;
type Pig is new Animal with …;

type Acc_Horse is access all Horse'Class;
type Acc_Pig is access all Pig;
```

An anonymous access type may be used as a component subtype in the declaration of an array type or object:

```
Napoleon, Snowball : Acc_Pig := …;
Boxer, Clover      : Acc_Horse := …;
Animal_Farm : constant array (Positive range <>) of
              access Animal'Class := (Napoleon,
                                      Snowball,
                                      Boxer,
                                      Clover);
```

Note the use of an anonymous access type for the component type of Animal_Farm. As is customary with anonymous access types, a value of a named access type can be implicitly converted to an anonymous access type, provided that the conversion is safe (i.e., goes towards the root of the type hierarchy). That's why the various animal names can be directly used in the aggregate. Contrast this situation with Ada 95, where a named access type would have to be used for the array component, and explicit conversions would be required in the aggregate.

Anonymous access types may also be used in record components, with the same implicit convertibility rules:

```
type Noah_S_Arch is
   record
      Stallion, Mare : access Horse;
      Boar, Sow      : access Pig;
   end record;
```

For the purpose of the accessibility rules, such components have the same accessibility level as that of the enclosing type. This is necessary to prevent dangling refer-

ences, and it means that from this perspective the anonymous access types behave as if they were implicitly declared immediately before the composite type.

Finally, it is also possible to use anonymous access types in object renaming declarations. This is useful to reference a value without changing its accessibility level. Consider the example:

```
procedure Feast (The_Arch : access Noah_S_Arch) is
   A_Pig : access Pig renames The_Arch.Sow;
begin
   Roast (A_Pig);
end Feast;
```

Here the accessibility level of The_Arch is that of the actual parameter. The renaming makes it possible to reference the component Sow in a manner that doesn't alter the accessibility level (in other words, the accessibility level of A_Pig is that of the actual parameter passed to Feast). Thus the accessibility level is preserved when passing A_Pig to Roast. Using a named access type for A_Pig would change the accessibility level, which may be problematic when executing the body of Roast.

4.10 Accidental Overloading When Overriding (AI 218)

It is possible in Ada 95 (and in other programming languages, e.g., Java, C++) for a typographic error to change overriding into overloading. When this happens, the dynamic behavior of a program won't be what the author intended, but the bug may be very hard to detect. Consider for instance:

```
with Ada.Finalization;
package Root is
   type Root_Type is new Ada.Finalization.Controlled
      with null record;
   -- Other primitive operations here.
   procedure Finalize (Object : in out Root_Type);
end Root;

with Root;
package Leaf is
   type Derived_Type is new Root.Root_Type
      with null record;
   -- Other primitive operations here.
   procedure Finalise (Object : in out Derived_Type);
end Leaf;
```

Here presumably the author of package Leaf intended to redefine the procedure Finalize to provide adequate finalization of objects of type Derived_Type. Unfortunately, she used the British spelling, so the declaration of Finalise (note the 's') does not override the inherited Finalize (with a 'z'). When objects of type Derived_Type are finalized, the code that is executed is that of Root.Finalize, not that of type Leaf.Finalise.

This is obviously a safety concern, and some programming languages (e.g., Eiffel, C#) provide mechanisms to ensure that a declaration is indeed an override. The ARG intends to provide a similar mechanism, although the details are still unclear. The ap-

proach which seems the most promising is that of introducing new syntax. An overriding declaration would have to include the word "overriding" (a non-reserved keyword) as in the following example:

```
overriding
procedure Finalise (Object : in out Derived_Type);
```

This declaration would actually result in an error, since Finalise doesn't override any subprogram (in particular, it doesn't override the parent type's Finalize), and the spelling error would be detected early, at compilation time.

There are a number of issues that are still unresolved, though, having to do with overriding in generics and with the interactions with private types, so a lot of work is still needed to consolidate this proposal.

4.11 Pragma Assert, Pre-Conditions and Post-Conditions (AIs 286 and 288)

Several Ada compilers support an Assert pragma, in largely the same form. As part of the Amendment work, the ARG intends to standardize this pragma, an associated package, and an associated configuration pragma for controlling the effect of the pragma on the generated code.

AI 286 defines pragma Assert as taking a boolean expression and optionally a message:

```
pragma Assert (Angle in 0.0 .. Pi / 2 or
               Angle in Pi .. 3 * Pi / 2,
               Message => "Angle out of range");
```

This pragma may appear anywhere, including in a declarative part. At execution, the boolean expression is evaluated, and if it returns False the exception Assertion_Error is raised with the indicated message. This exception is declared in a new predefined package, Ada.Assertions:

```
package Ada.Assertions is
   pragma Pure (Ada.Assertions);
   Assertion_Error : exception;
end Ada.Assertions;
```

In practice, it is also useful to be able to enable or disable assertion checking on an entire program or on a collection of units. In order to help with this usage model, AI 286 also defines a configuration pragma, Assertion_Policy. Thus, the pragma:

```
pragma Assertion_Policy (Ignore);
```

disables all assertion checking in the entire environment or in specific units, while the pragma:

```
pragma Assertion_Policy (Check);
```

enables normal assertion checking.

On a related topic, the ARG has studied the possibility of improving the support of the "programming by contract" model, in a manner similar to what Eiffel provides. This would be done by expressing pre- and post-conditions for subprograms, and in-

variants for types and packages. AI 288 defines a number of pragmas to that effect. However, this proposal is not yet mature, and its details are very much in flux.

4.12 Unchecked Unions: Variant Records with No Run-Time Discriminant (AI 216)

Ada does not include a mechanism for mapping C unions to Ada data structures. At the time of the design of Ada 95, it was thought that using Unchecked_Conversion to obtain the effect of unions was satisfactory. However, easy interfacing with C unions is important enough that several compilers have defined a method to support it. The ARG has decided to standardize this interfacing technique.

AI 216 defines a new representation pragma, Unchecked_Union, which may be applied to a discriminated record type with variants to indicate that the discriminant must not exist at run-time. For example, consider the following C type, which could represent an entry in the symbol table of a compiler, where a symbol could be either an object or a package:

```
struct sym {
   int id;
   char *name;
   union {
      struct {
         struct sym *obj_type;
         int obj_val_if_known;
      } obj;
      struct {
         struct sym *pkg_first_component;
         int pkg_num_components;
      } pkg;
   } u;
};
```

This data structure maps to the following unchecked union type in Ada:

```
type Sym;
type Sym_Ptr is access Sym;
type Sym_Kind_Enum is (Obj_Kind, Pkg_Kind);
type Sym (Kind : Sym_Kind_Enum :=
                 Sym_Kind_Enum'First) is
   record
      Id : Integer := Assign_Id (Kind);
      Name : C_Ptr;
      case Kind is
         when Obj_Kind =>
            Obj_Type : Sym_Ptr;
            Obj_Val_If_Known : Integer := -1;
         when Pkg_Kind =>
            Pkg_First_Component : Sym_Ptr;
            Pkg_Num_Components  : Integer := 0;
      end case;
   end record;
pragma Unchecked_Union (Sym);
```

Because of the presence of the pragma, the discriminant Kind does not exist at run-time. This means that the application has to be able to determine from the context if a symbol is a package or an object. Those operations which would normally need to access the discriminant at run-time (like membership test, stream attributes, conversion to non-unchecked union types, etc.) raise Program_Error. Note that representation clauses may be given for an unchecked union type, but it is obviously illegal to give a component clause for a discriminant.

4.13 Directory Operations (AI 248)

Most modern operating systems contain some sort of tree-structured file system. Many applications need to manage these file systems (by creating and removing directories, searching for files, and the like). Many Ada 95 compilers provide access to these operations, but their implementation-defined packages differ in many ways, making portable programs impossible.

The POSIX libraries provide operations for doing this, but these libraries usually are available only on POSIX systems, leaving out many popular operating systems including MS-DOS®, most flavors of Windows®, and even Linux.

Therefore, the ARG has decided to standardize a minimum set of capabilities to access directories and files. The purpose is not to provide interfaces to of all the features of the underlying file system, but rather to define a common set of interfaces that makes it possible to write programs that can easily be ported on different operating systems. This is similar in spirit to the definition of Ada.Command_Line in Ada 95.

AI 248 defines a new predefined package, Ada.Directories, with operations to:

- Query and set the current directory;
- Create and delete a directory;
- Delete a file or a directory or an entire directory tree;
- Copy a file and rename a file or a directory;
- Decompose file and directory paths into enclosing directory, simple name, and extension;
- Check the existence, and query the size and modification time of a file;
- Iterate over the files and directories contained in a given directory.

It is intended that implementations will add operating-system-specific children of Ada.Directories to give access to functionalities that are not covered by this unit (e.g., file protections, encryption, sparse files, etc.).

5 Conclusion

This paper has given an overview of the standardization process for the 2005 revision of Ada, and it has presented those proposals that are reasonably stable and mature.

The ARG has also studied (and is still studying) a large number of other topics including: improving formal packages; allowing access-to-constant parameters; improv-

ing visibility rules for dispatching operations; and supporting notification of task termination. Some of these ideas seem promising, other might be disruptive, and so it's too early to tell which ones will make it into the Amendment and which ones will not.

The ARG's goal is to make Ada more powerful, safer, more appealing to its users, without imposing unreasonable implementation or training costs. Whatever the details of the changes that ultimately make it into the Amendment, we anticipate that 2005 will be a good vintage.

References

1. S. T. Taft, R. A. Duff, R. L. Brukardt, E. Ploedereder, Consolidated Ada Reference Manual, LNCS 2219, Springer-Verlag.
2. A. Burns, "The Ravenscar Profile", ACM Ada Letters, Vol. XIX, No. 4, pp. 49-52, December 1999.