Ada 2005 – Ready to Roll

Robert Dewar
CEO AdaCore

Ada is at this stage a fairly old language, having been first standardized over twenty years ago. In that time, it has been extensively used, particularly in large critical programs where, to quote Ed Harris in a well known movie, "failure is not an option". That's not at all surprising because Ada from its inception has been designed with this kind of reliability in mind.

Since Ada is a fairly old language, one often meets people who think it is obsolete. Basically they assume that anything that was not invented yesterday must be out of date by now. Well if they have in mind the original Ada 83, it is indeed true that this language is obsolete, though like many obsolete languages, that does not mean it cannot be effectively used. Why is it obsolete? Not because no one is using Ada (many critical new applications are being programmed in Ada), but rather because it was superceded ten years ago by Ada 95, which was at the same time, a completely new language with many new features, and a smooth upwards-compatible transition for Ada 83 users.

Well history repeats itself, and Ada 95 is also about to become obsolete, with the introduction of what we are calling Ada 2005 for now. There is no really official name until the new standard is issued. The best guess is that this will happen early in Ada 2006, at which time the official name will be simply Ada (Ada always refers to the latest standardized version, we use Ada 83, Ada 95 etc to distinguish for historical reasons). For now, we choose the name Ada 2005, since it is this year that the features of the new revision are finally frozen to the point that major implementation work can begin, and indeed at least one major commercial implementation of Ada (GNAT Pro from AdaCore) already implements many of the most important features of Ada 2005, and other Ada implementations can be expected to follow.

Language design is definitely a field in which we continue to refine our ideas, and of course we learn by experience. Ada 95 was the first internationally standardized object oriented language, and we have learned a lot in ten years, so with the Ada 2005 revision, we have taken the opportunity to bring the Ada definition up to the state of the art in programming language design, while preserving a remarkably high level of compatibility with previous versions of Ada, so that existing legacy code will not need extensive changes and indeed in most cases will require no changes at all.

Ada 2005 is a new language, but the fundamental design principles on which it is based are not new. Why does Ada stand out from other languages? The most important answer is that it was designed with large critical, high reliability, programs in mind. If you have code that controls avionics in a commercial jet (for example the Boeing 777), a critical component of the space station (the "Canadian space arm"), medical equipment (from JEOL), air traffic control (Eurospace in Europe) or a critical ejection seat mechanism in a jet fighter [need reference], reliability is not just desirable, it is mandatory. Ada is

designed from the start with this kind of application in mind, and the new Ada 2005 revision continues to embody this fundamental viewpoint of safety and reliability.

When we are programming in these kind of environments, it is not acceptable to have our application bomb out with a message asking if we want to send a report to Microsoft. We demand absolute reliability, and we need to be able to detect errors as soon as possible. Some programmers find Ada a nuisance because it is so picky about what is acceptable. A common experience among Ada programmers is that it is sometimes hard work to get the compiler to accept code but once accepted it is far more likely to work. Contrast this with C++, where the compiler will accept all sorts of stuff, and then generate code that badly misbehaves. An interesting aspect of this contrast is that C++ programmers cannot imagine surviving without a debugger. In the Ada world, debuggers are certainly used, and excellent Ada knowledgable debuggers are available. However, many Ada programmers find they can manage most or all of the time without ever using the debugger.

As another example of this orientation, consider the statement:

```
A = A + 1
```

which increments the value of A. Let's suppose that A is the largest integer that can be represented, resulting in overflow. What happens? Well in C++, the result is undefined, which means that absolutely anything might happen. An implementation that activated the self destruct on the rocket in this situation would be perfectly conforming. In Java, you silently get the most negative number possible as the result, which hardly bodes well for future reasonable behavior, given that the programmer expects A to be positive after adding 1 to it. In Ada, this raises an exception which can be caught by the program, and appropriate corrective action taken.

One other aspect of Ada that is critically important, and again Ada 2005 has maintained this design viewpoint, is that we always favor the reader over the writer. We don't care if the coder has to work a bit harder or write a bit more, if this makes it easier for the reader. The reason for this decision seems obvious. In practice code is written once, but read, maintained, modified, and reused many times. Maintenance programming is difficult, expensive, and hazardous. Anything that makes this easier is welcome, even at the expense of making the original coder work a bit harder.

Much of what we say here applies to all versions of Ada, but there are many important new features in Ada 2005 which will make the language even more convenient for the construction of large scale reliable programs.

New Object Oriented Features

A conscious decision in Ada 95 was not to implement multiple inheritance at all. This decision was taken because we really did not understand how to do this without creating a confusing mess (the problems surrounding virtual base classes in C++ are symptomatic

of this confusion). But more recently, the notion of interfaces has been developed as an effective alternative which gives the power of interfacing to multiple abstractions without the confusion of full multiple inheritance.
Java implements a version of this interface notion, and Ada 2005 builds on these ideas to create a new and powerful form of the interface abstraction, which also extends to the unique Ada notions of task and concurrent object, maintaining the important design principle that concurrency is a first class citizen in Ada, not something that is glued on as in C++, or implemented at a low level as in Java.

Other important object oriented features allow use of the prefix notation where appropriate, and control over overriding when extending types (we discuss these points in more detail later on).

Simplified Tasking Model

The tasking model of Ada 95 is extremely powerful, but it was always recognized that in some circumstances, particularly in the case of safety critical applications, it would be appropriate to choose a subset of these facilities which could be certified without unreasonable effort. The Ravenscar model of tasking is now a well defined subset of this kind in Ada 2005, and all Ada 2005 implementations must provide support for it. The intention is that not only will they support it, but in appropriate environments, notably embedded environments, efficient implementations of the Ravenscar tasking model will be supplied by Ada 2005 implementations.

Internationalization

In an increasingly international world, the issue of properly handling the diverse character sets of the world both in data within a program, and in the text of the source program itself becomes an important issue that cannot be ignored. Ada 2005 is the first language to fully address this issue. The Ada 2005 definition takes full advantage of the extensive work done in defining the latest version of the ISO 10646 and Unicode standards. The full range of characters defined by the standards is available for use in data that is read and written by the program, string literals within the program, and identifiers in the source. This definition is rather complex since the languages of the world are extremely diverse. Just a taste of the complexity may be seen in the fact that in Turkish, the upper case equivalent of lower case I (with a dot) is upper case I (with a dot, unfortunately not available in the character set of this word processor). The characters upper case I with a dot and upper case I without a dot are quite different in Turkish, and so are the corresponding lower case letters.

It is remarkably tricky to get everything right in this area, but this has been achieved in Ada 2005, which allows full flexibility of character sets, while preserving the notions of case equivalence, readability of identifiers, and portability that are important elements of the Ada language.

New Library Facilities

In modern programming environments, the availability of libraries of units providing various useful functions can be an important aid to efficient and effective programming. An important caveat is that these libraries must be designed with the same degree of care as the language itself. Many languages suffer from a "thrown-together" jumble pile of miscellaneous library units. The Ada design has sought to avoid this trap, while still providing an effective suite of library capabilities.

A substantial addition in Ada 2005 is a large set of new library units. In particular, the so-called "container" library provides a set of container facilities (sets, maps, sequences, hash tables etc). These have been carefully reviewed and designed by the Ada 2005 committee, so that they fit smoothly into the language and can be implemented reliably and efficiently.

Other new library facilities include standardized access to directories, new time and calendar functions, and additions and improvements to many existing library units.

Object Oriented Programming and Safety Critical Requirements

Object oriented programming is a term that covers a wide variety of ideas and features. At one end we have traditional object oriented design (in which a problem is modeled as a set of objects with message passing). Such designs can perfectly well be programmed in languages with no "object oriented features", and do not necessarily raise any special issues in the safety critical arena. At the other end, we have the features that traditionally appear in what are known as "object oriented languages", namely type extension, inheritance and dynamic dispatching.

Ada 95 was the first internationally standardized object oriented programming language in this latter sense, and so it is not unexpected that Ada 2005 makes substantial advances in this area, particularly with respect to safety critical requirements. The situation is that type extension and inheritance do not cause any problems, but dynamic dispatching is indeed worrisome, and there is no general agreement on how to handle dynamic dispatching from a certification point of view [reference? Ask Cyrille]. One conservative approach is to allow type extension and inheritance but to avoid dynamic dispatching. Ada 2005 facilitates this approach in a number of ways. First there is a sharp distinction between the normal use of tagged types, and their corresponding class types. If class types are avoided, then dynamic dispatching never occurs, and still it is possible to make full use of inheritance and type extension, making code reuse much easier. Second, this can be enforced by use of a language defined restriction (No_Dispatch). Finally, a feature new with Ada 2005 allows very careful control over inheritance, by allowing each operation to declare explicitly whether it is intended to inherit, and the compiler checks that the intention is met (this avoids accidentally confusing Initialize and Initialise for example, a well known hazard in OOP languages).

The basic notion of OOP features in Ada 2005 is much more powerful than in comparable other languages. The same level of power is present (including multiple

inheritance, which is done with interfaces that will be at least a little familiar to Java programmers, but avoiding the complexity of MI as defined in C++). However, the model recognizes that OOP features can be used for far more than traditional object oriented design. For example, if we have an arithmetic package that declares an addition operator on some specialized arithmetic type, we can write in Ada 2005:

```
X := Y + Z;
```

Where + invokes the specialized method appropriate to the type of Y and Z. In other object oriented languages, this has to be modeled as one of the objects passing a message to the other saying something like "please add me to you", which is really rather strained. If we had to write, as in other languages:

```
X := Y."+"(type_convert(Z));
```

then we have lost a lot of clarity, and clarity is, as previously discussed a non-negotiable requirement in Ada programs. At the same time it is indeed the case that if you are using a message passing paradigm, then the insistence on using functional forms in Ada 95 proved annoying. This has been corrected in Ada 2005 by allowing prefix notation to be used optionally where it is appropriate, so this means that the two following forms are equivalent:

```
X := Name (Arg1, Arg2);
X := Arg1.Name (Arg2);
```

Which choice is better depends on the abstract meaning. By giving the programmer the choice between these two forms, we allow the thought to be expressed in the clearest possible manner. As usual, we have in mind the convenience of the maintenance programmer and code reuser.

Given that we have now ten years of experience with the use of a standardized object oriented language, we have learned from this experience, and Ada 2005 incorporates many small and useful additions in this area that are designed to make the code clearer to read. For example, we found that the Ada 95 design resulted in some cases in excessive numbers of type conversions, so the language has been amended, and these "junk" conversions have been eliminated.

It is clear that the programmers writing the next generation of safety critical programs will want to take advantage of the powerful notions of object oriented programming. It simply won't be acceptable to forbid this usage. At the same time, we have to be very careful in controlling what is permitted, and how it is tested. Ada 2005 is ideally suited as the vehicle for exploiting what is safe in this area, while avoiding what is dangerous.

Summary of the new Ada

In the sections above we have outlined some of the major features of Ada 2005. We don't have room here to list all the other features that are added. Individually they are small and do not affect compatibility with existing Ada programs. Together they add up to a new era of high reliability computing using this new and improved version of Ada.

Use of Ada in Embedded Environments

Although Ada is at home in many different computing environments, including even classical mainframe computing, the place where Ada really shines is its use in embedded applications. There are two reasons for this. First, Ada was designed with embedded applications in mind from the start. For example, the use of representation clauses, which have been extended and made more powerful in Ada 2005, allows close mapping of data structures to the hardware, and the built in concurrency can be used to map handling of multi-tasking at the hardware level. Second, many embedded applications are precisely of the high reliability or safety critical level where a language designed for maximum safety really shines.

Another important feature of Ada is that it maps very well to the typical embedded operating systems used in many embedded applications. For example, the Boeing 787 uses the WRS Arinc 653 system. This safety critical kernel was designed with Ada in mind. Nor surprisingly Ada implementations are available to match this capability, and the 787 will have significant amounts of Ada code aboard.

What about the Tools?

A language no matter how well designed, and its compiler, no matter how well implemented are only part of the story. These days, we expect a complete integrated development environment to be available, allowing for convenient code/test/debug/fix cycles, and also allowing easy access to a powerful tool suite.

Ada 2005 will not lag behind in this department. Several powerful Ada oriented integrated development environments are available, including the GPS (GNAT Programming Studio) from AdaCore, and Workbench from Wind River Systems. Furthermore at least three Ada companies are developing sophisticated Ada oriented plugins for the Eclipse environment.

In each of these environments, a powerful set of tools, including such functions as intelligent builds, automatic documentation, code metrics, static analysis tools, configuration management tools is available, and it is also possible through simple scripting languages to integrate new tools and capabilities.

Summary

Far from disappearing, Ada is alive and well and growing. The new Ada 2005 version will help maintain Ada's position as the premier language choice when life depends on

the reliability of code, as is increasing the case in our modern society, where embedded devices are all around us.