Ada 2005 for High-Integrity Systems

José F. Ruiz

AdaCore
8 rue de Milan
75009 Paris, France

ruiz@adacore.com

Abstract

The forthcoming Ada 2005 standard has been enhanced to better address the needs of the real-time and high-integrity communities. This new standard introduces new restriction identifiers that can be used to define highly efficient, simple, and predictable run-time profiles. Among others, this language revision will standardize the Ravenscar profile, new scheduling policies, and will include execution time clocks and timers. Flexible object-oriented features are also supported without compromising performance or safety.

1 Introduction

For the development of safety-critical software, the choice of programming language makes a significant difference in meeting the requirements of exacting safety standards and, ultimately, high-reliability applications.

The Ada language was first introduced in 1983 (ISO 1983). Used primarily for large-scale safety and security critical projects, and embedded systems in particular, where reliability and efficiency are essential, Ada experienced its last major revision in 1995 (ISO 1995), making it the first internationally standardized object-oriented language. The latest revision (Ada 2005) responds to requests for features in the areas of multiple interface inheritance, real-time profiles, flexible task-dispatching policies, and a unification of concurrency and object-oriented features.

One of the most important achievements of Ada 2005 is the standardization of the Ravenscar restricted tasking profile. This profile defines a subset of the tasking features of Ada which is amenable to static analysis for high integrity system certification, and that can be supported by a small, reliable run-time system. This profile is founded on state-of-the-art, deterministic concurrency constructs that are adequate for constructing most types of real-time software.

Measuring and limiting the execution time of tasks is also possible in Ada 2005 by using execution time clocks and timers. This functionality is equivalent to the

execution time monitoring existing in the real-time extension to POSIX (IEEE 2003), allowing the implementation of flexible real-time scheduling algorithms, such as the sporadic server in fixed priority systems, or the constant bandwidth server in dynamic priority systems.

Timing events are also provided as an effective and efficient to execute user-defined time-triggered procedures without the need to use a task or a delay statement.

There have been major improvements to the scheduling and task dispatching mechanisms with the addition of further standard pragmas, policies, and packages which facilitate many different mechanisms such as non-preemption within priorities, timeslicing, and dynamic priority dispatching. Moreover, it is possible to mix different policies according to priority ranges within a partition.

The following sections will describe the advantages of using Ada for developing embedded real-time high-integrity systems, paying special attention to the new features that will be available in the forthcoming Ada 2005 standard.

2 Software engineering with Ada

The general design philosophy of the language promotes sound software engineering techniques basing on its considerable expressive power and high abstraction level features.

The original Ada 83 design introduced the package construct, a feature that supports encapsulation (information hiding) and modularization, and that allows the developer to control the namespace that is accessible within a given compilation unit, hence reducing data coupling. Ada 95 introduced the concept of child units, adding considerably flexibility and easing the design of very large systems. Packages provide strict separation of specification from implementation, and allow the structuring of code into a hierarchical set of components with strict control over visibility of encapsulated state data and methods.

One important capability of the child unit mechanism is that it allows developers to write test programs that can access encapsulated state data that is inaccessible to normal client code. This simplifies the job of meeting coverage analysis requirements from safety standards such as DO-178B (RTCA 1992), without compromising the need to have state data hidden.

Generics are a powerful mechanism for constructing large-scale programs through the parameterization of program units. The use of generics enhances program reliability by means of facilitating reuse, easing maintenance, reducing source code size, and helping avoid human replication error.

Ada 95 introduced direct support for object-oriented programming: encapsulation (as just noted), objects (entities that have state and operations), classes (abstractions of objects), inheritance, polymorphism, and dynamic binding.

Ada tasking provides a natural and powerful abstraction mechanism for decoupling application activities, including the functionality for sharing resources, communicating, and synchronizing.

3 Ada for embedded applications

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. Additionally, many embedded applications require high reliability or are safety-critical, which is where a language designed for maximum safety really shines.

The Ada standard includes a normative annex which specifies additional capabilities provided for low-level programming (ISO 1995, Annex C). It allows access to hardware-specific features, such as:

- Insertion of assembly and intrinsic subprograms. Intrinsic subprograms are built-in to the compiler provided for convenient access to any machine operations that provide special capabilities or efficiency and that are not otherwise available through the language constructs. Examples of such instructions include atomic read-modify-write operations, standard numeric functions, string manipulation operations, vector operations, direct operations on I/O ports, etc.
- Representation clauses for specifying the desired address, size, alignment, and layout of data in memory.
- Shared variable control. Read and update operations can be forced to either be performed directly to memory, or in a indivisible (atomic) manner.
- Interrupt support. There is a language-defined model for hardware interrupts which includes the mechanisms for handling interrupts.
- Storage management. Specific storage pools can be specified with user-defined managers that may be placed in specific memory regions. They may be suitable for real-time systems because they can be made predictable.

Another important feature of Ada is that its functionality, notably its tasking capabilities, maps very well to the typical embedded operating systems used in many applications.

4 Ada for high-integrity applications

Ada is the language of choice for many high-integrity systems due to its careful design and the existence of clear guidelines for building high integrity systems (ISO 2000, Burns, Dobbing & Vardanega 2003).

Fitting its commitment to safety and reliability, a formal validation process exists based on an ISO (International Standards Organization) standard (ISO 1999). Ada is the only language for which such a validation standard exists. An Ada Conformity Assessment Test Suite (ACATS) (ACAA 2005) has been developed for this conformity testing, which exercises both the compiler and the run-time system.

The use of a standardized language (ISO 1999) ensures that your program will behave as you want (as it is designed to) even when changing target platforms or compilers. The effect of a program can be predicted from the language definition with few implementation dependencies of interactions among language features. The semantics of Ada programs are well defined even in error situations. The Ada standard includes a normative annex which specifies additional capabilities provided for systems that are safety critical or have security constraints (ISO 1995, Annex H).

When writing high reliability software, the full Ada language is inappropriate since the generality and flexibility may interfere with traceability and certification requirements. Ada addresses this issue by supplying configuration directives (that may restrict individual features or define a complete set of restrictions) that allows you to constrain the language features to a well-defined subset that facilitate analysis and safety, and avoids error prone or hard to analyze features. The ISO 15942 technical report (ISO 2000) contains a detailed analysis of the different Ada features with respect to their suitability for different verification techniques. The use of restricted profiles and restrictions also allows the compiler to remove unnecessary run-time support, simplifying the certification process and preventing the inclusion of inactive code in the final application.

One of the most interesting subsets for high-integrity systems is the Ravenscar profile, a collection of concurrency features that are powerful enough for real-time programming but simple enough to make certification practical. Another notable example is SPARK (Barnes 2003) that includes Ada constructs regarded as essential for the construction of complex software, but removes all the features that may jeopardize the requirements of verifiability, bounded space and time, and minimal run-time system.

Apart for the advantages derived from the high abstraction level provided by the language (encapsulation, data abstraction, reusability, tasking, etc.), there are many others features in the language that promote safety and reliability. Ada code is very readable, making code maintenance easier and simplifying certification steps, including peer review and walkthroughs. Strong typing ensure that most errors are detected statically at compile time, and many remaining errors are automatically detected at execution.

Access types in Ada have been designed in a way to prevent the occurrence of dangling references because they can never designate objects that have gone out of scope. Users can also further restrict the use of allocators and deallocators through appropriate restrictions.

Ada provides an exception mechanism for detecting and responding to exceptional run-time conditions in a controlled manner, providing well-defined semantics even under error conditions. It allows residual errors to be detected and handled, so the exception features are potentially a key part of a language for high-integrity applications (Motet, Marpinard & Geffroy 1996). Its use makes verification more difficult, unless restrictive strategies (ISO 2000) are used which simplify the verification process.

Ada 2005 contains determinism and hazard mitigation issues relating to task activation and interrupt handler execution semantics, in response to certification con-

cerns about potential race conditions that could occur due to tasks being activated and interrupt handlers being executed prior to completion of the library-level elaboration code. A new configuration pragma has been added (ARG 2005*d*) for guaranteeing the atomicity of program elaboration, that is, no interrupts are delivered and task activations are deferred until the completion of all library-level elaboration code. This eliminates all hazards that relate to tasks and interrupt handlers accessing global data prior to it having been elaborated, without having to resort to potentially complex elaboration order control.

Another major hazard in high-integrity systems, tasks terminating silently, has been addressed in Ada 2005 with a new mechanism for setting user-defined handlers which are executed when tasks are about to terminate. These procedures are invoked when tasks are about to terminate (either normally, as a result of an unhandled exception, or due to abort), allowing controlled responses at run time and also logging these events for post-mortem analysis.

5 Ada for real-time applications

Concurrency is a core part of the language, and there is a normative annex intended for real-time systems software (ISO 1995, Annex D) that supports sound real-time development techniques, such as Rate Monotonic Analysis (Liu & Layland 1973), Response Time Analysis (RTA) (Joseph & Pandya 1986), and some others introduced in the Ada 2005 revision that will be described later.

Ada provides well-defined semantics for scheduling, avoiding the disadvantages associated with the use of low-level constructions for task handling and synchronization. Task cooperate using synchronous message passing (rendezvous) and safe and efficient data-oriented communication and synchronization through protected objects.

Asynchronous capabilities are also very important for some real-time applications, and they are supported with the following mechanisms:

- Asynchronous Transfer of Control (ATC) is a mechanism that allows the execution of an abortable part to be cancelled by a triggering event (time event or another task), in which case an optional sequence of code can be executed after the abortable part is left.
- Preemptive task abortion can trigger asynchronously the termination of one or more target tasks.
- Asynchronous task control is a simple and efficient capability to suspend and resume the execution of another task.
- Asynchronous external events are modelled by interrupts, a language-defined class of events that are detected by the hardware or the system software.

A high-resolution monotonic clock together with support for both absolute and relative delays are also part of the Ada standard, which defines minimum requirements in terms of range and accuracy.

6 The Ravenscar profile

As the functionality and complexity of embedded software increases, more attention is being devoted to high level, abstract development methods. The Ada tasking model provides concurrency as a means of decoupling application activities, and hence making software easier to design and test (Vardanega & van Katwijk 1999).

The tasking model in Ada 95 is extremely powerful, but it has always been recognized that, in the case of high-integrity systems, it is appropriate to choose a subset of these facilities because accurate timing analysis is difficult to achieve. Advances in real-time systems timing analysis methods have paved the way to reliable tasking in Ada. Accurate analysis of real-time behavior is possible given a careful choice of scheduling/dispatching method together with suitable restrictions on the interactions allowed between tasks.

The Ravenscar profile (ARG 2005*f*) is a subset of Ada tasking that provides the basis for the implementation of deterministic and time analyzable applications. This subset is amenable to static analysis for high integrity system certification, and can be supported by a small, reliable run-time system. This profile is founded on state-of-the-art, deterministic concurrency constructs that are adequate for constructing most types of real-time software (Burns et al. 2003). Major benefits of this model are:

- Improved memory and execution time efficiency, by removing high overhead or complex features.
- Increased reliability and predictability, by removing non-deterministic and non analyzable features.
- Reduced certification cost by removing complex features of the language, thus simplifying the generation of proof of predictability, reliability, and safety.

The profile is based on a computation model similar to the one proposed by Vardanega (Vardanega 1998), which is based on the HRT-HOOD method (Burns & Wellings 1995), that includes the following features:

- A single processor.
- A fixed number of tasks.
- A single invocation event per task (either time-triggered or event-triggered tasks).
- Task interaction only by means of shared data (protected objects) with mutually exclusive access.

Constructions that are difficult to analyze, such as dynamic tasks and protected objects, task entries, dynamic priorities, select statements, asynchronous transfer of control, relative delays, or calendar clock, are forbidden. It allows memory usage and execution to be deterministic.

The concurrency model promoted by the Ravenscar Profile is consistent with the use of tools that allow the static properties of programs to be verified. Potential verification techniques include information flow analysis, schedulability analysis, execution-order analysis and model checking.

The Ravenscar profile will be part of the Ada 2005 standard, so compiler vendors must implement 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 also be supplied.

7 Scheduling and dispatching policies

An important area of increased flexibility in Ada 2005 is that of task dispatching policies. In Ada 95, the only predefined policy is fixed-priority preemptive scheduling, although other policies are permitted. Ada 2005 provides further pragmas, policies, and packages which facilitate many different mechanisms such as non-preemption within priorities (ARG 2005*c*), round robin using timeslicing (ARG 2005*e*), and Earliest Deadline First (EDF) policy (ARG 2005*g*). Moreover, it is possible to mix different policies according to priority levels within a partition.

Time sharing the processor using round robin scheduling is adequate for non-real-time systems, and also in some soft real-time systems requiring a level of fairness. Many operating systems, including those compliant with the POSIX real-time scheduling model, support this scheduling policy that ensures that if there are multiple tasks at the same priority one of them will not monopolize the processor.

In order to reduce non-determinism and to increase the effectiveness of testing, non-preemptive execution is sometimes desirable (Burns 2001). The standard way of implementing many high-integrity applications is with a cyclic executive (Baker & Shaw 1989). Using this technique a sequence of procedures is called within a defined time interval. Each procedure runs to completion and there is no concept of preemption. Data is passed from one procedure to another via shared variables and no synchronization constraints are needed, since the procedures never run concurrently. The major disadvantage with non-preemption is that it will usually (although not always) lead to reduced schedulability.

Ada 2005 supports the notion of deadlines (the most important concept in real-time systems) via a predefined task attribute. The deadline of a task is an indication of the urgency of the task. EDF scheduling allocate the processor to the task with the earliest deadline. EDF has the advantage that higher levels of resource utilization are possible, although it is less predictable, compared to fixed-priority scheduling, in case of overload situations.

8 Execution time monitoring and control

Monitoring and control execution time is important for many real-time systems. Ada 2005 provides an additional timing mechanism (ARG 2005*a*, ARG 2005*b*) which allows for:

- monitoring execution time of individual tasks,
- defining and enabling timers and establishing a handler which is called by the run-time system when the execution time of the task reaches a given value, and
- defining a execution budget to be shared among several tasks, providing means whereby action can be taken when the budget expires.

This functionality is easily supported on top of operating systems compliant to the real-time extensions to POSIX (IEEE 2003), that has recently incorporated support for execution time monitoring and budgeting.

Monitoring CPU usage of individual tasks can be used to detect at run time an excessive consumption of computational resources, which are usually caused by either software errors or errors made in the computation of worst-case execution times.

Schedulability analysis are based on the assumption that the execution time of each task can be accurately estimated. Measurement is always difficult, because, with effects like cache misses, pipelined and superscalar processor architectures, the execution time is highly unpredictable. Run-time monitoring of processor usage permits detecting and responding to wrong estimations in a controlled manner.

CPU clocks and timers are also a key requirement for implementing some modern real-time scheduling policies which need to perform scheduling actions when a certain amount of execution time has been consumed. Providing common CPU budgets to groups of tasks is the basic support for implementing aperiodic servers, such as sporadic servers and deferrable servers (Sprunt, Sha & Lehoczky 1989) in fixed priority systems, or the constant bandwidth server (Ghazalie & Baker 1995) in EDF-scheduled systems.

9 Timing events

Timing events (ARG 2005*h*) allow for a handler to be executed at a future point in time in a efficient way, as it is a stand-alone timer which is execute directly in the context of the interrupt handler (it does not need a server task).

The use of timing events may reduce the number of tasks in a program, and hence reduce the overheads of context switching. It provides an effective solution for programming short time-triggered procedures, and for implementing some specific scheduling algorithms, such as those used for imprecise computation (Liu, Lin, Shih, Chuang-Shi, Chung & Zhao 1991). Imprecise computation increase the utilization and effectiveness of real-time applications by means of structuring tasks into two phases (one mandatory and one optional). Scheduling algorithms that try to maximize the likelihood that optional parts are completed typically require changing asynchronously the priority of a task, which can be implemented elegant and efficiently with timing events.

10 Object-oriented programming

Object-oriented programming is a term that covers a broad spectrum 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 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.

Programmers writing high-integrity systems want to take advantage of the powerful notions of object-oriented programming, and work is being done in the direction of providing guidelines for certifying object-oriented applications (FAA 2004). Ada 2005 is ideally suited as the vehicle for exploiting what is safe in this area, while avoiding what is dangerous.

Given Ada's emphasis on high-integrity applications, Ada 2005 directly addresses the use of object-oriented methods within the constraints of these kinds of systems. Type extension and inheritance do not cause any problems, but dynamic dispatching is worrisome, and there is no general agreement on how to handle dynamic dispatching, where the actual flow of controls is not known statically but at run time, from a certification perspective (based on knowing the flow of control statically so it can be tested). 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 inheritance (tagged types) and dynamic dispatching (their associated class-wide types). In Ada, methods are statically bound by default. If class-wide types are avoided, then dynamic dispatching never occurs, and it is still possible to make full use of inheritance and type extension, thus facilitating code reuse. Second, this can be enforced by use of a language defined restriction (*No_Dispatch*). Finally, Ada 2005 offers very fine-grained 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 object-oriented languages).

A conscious decision was made in the design of Ada 95 to not implement general multiple inheritance, because the complexities introduced to the language appeared to overwhelm the benefits. Idiomatic usage of Ada 95 object-oriented facilities still provided the ability to implement multiple inheritance at the application level through such features as access discriminants and generic units with class-wide formal parameters. But more recently, the notion of interfaces (or roles) has been developed as an effective alternative that gives the power of interfacing to multiple abstractions without the additional complexity of full multiple inheritance. Java introduced the idea of interfaces, and Ada 2005 builds on the concept 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.

11 Conclusions

Ada's reliability has been field-proven for decades, even as the language evolves through real world innovation. The latest Ada 2005 responds to requests for features in the areas of multiple interface inheritance, real-time profiles, flexible task-dispatching policies, and a unification of concurrency and object-oriented features.

Safe tasking is promoted by the Ravenscar profile, which defines a deterministic and certifiable tasking subset, providing the high-level abstraction and expressive power needed for making software easy to design and test. Major hazards related to tasks terminating silently and potential race conditions at elaboration time have been addressed by new mechanisms added to Ada 2005.

The new language revision constitutes also the reference framework for high-integrity object-oriented programming, supporting powerful and flexible object-oriented features while avoiding those that jeopardize system certification.

Ada continues to be the reference language for high-integrity systems, providing high-level abstractions without compromising performance or safety.

References

ACAA (2005), *Ada Conformity Assessment Test Suite (ACATS)*, ACAA. Available at `http://www.ada-auth.org/acats.html`.

ARG (2005*a*), Execution-time clocks, Technical report, ISO/IEC/JTC1/SC22/WG9. Available at `http://www.ada-auth.org/cgi-bin/cvsweb.cgi/AIs/AI-00307.TXT`.

ARG (2005*b*), Group execution-time budgets, Technical report, ISO/IEC/JTC1/SC22/WG9. Available at `http://www.ada-auth.org/cgi-bin/cvsweb.cgi/AIs/AI-00354.TXT`.

ARG (2005*c*), Non-preemptive dispatching, Technical report, ISO/IEC/JTC1/SC22/WG9. Available at `http://www.ada-auth.org/cgi-bin/cvsweb.cgi/AIs/AI-00298.TXT`.

ARG (2005*d*), Partition elaboration policy for high-integrity systems, Technical report, ISO/IEC/JTC1/SC22/WG9. Available at `http://www.ada-auth.org/cgi-bin/cvsweb.cgi/AIs/AI-00265.TXT`.

ARG (2005*e*), Priority specific dispatching including round robin, Technical report, ISO/IEC/JTC1/SC22/WG9. Available at `http://www.ada-auth.org/cgi-bin/cvsweb.cgi/AIs/AI-00355.TXT`.

ARG (2005*f*), Ravenscar profile for high-integrity systems, Technical report, ISO/IEC/JTC1/SC22/WG9. Available at `http://www.ada-auth.org/cgi-bin/cvsweb.cgi/AIs/AI-00249.TXT`.

ARG (2005*g*), Support for deadlines and earliest deadline first scheduling, Technical report, ISO/IEC/JTC1/SC22/WG9. Available at `http://www.ada-auth.org/cgi-bin/cvsweb.cgi/AIs/AI-00357.TXT`.

ARG (2005*h*), Timing events, Technical report, ISO/IEC/JTC1/SC22/WG9. Available at `http://www.ada-auth.org/cgi-bin/cvsweb.cgi/AIs/AI-00297.TXT`.

Baker, T. & Shaw, A. (1989), 'The cyclic executive model and Ada', *Real-Time Systems*.

Barnes, J. (2003), *High Integrity Software. The SPARK Approach to Safety and Security*, Addison Wesley.

Burns, A. (2001), Defining new non-preemptive dispatching and locking policies for Ada, *in* D. Craeynest & A. Strohmeier, eds, 'Reliable Software Technologies — Ada-Europe 2001', number 2043 *in* 'Lecture Notes in Computer Science', Springer-Verlag, pp. 328–336.

Burns, A. & Wellings, A. (1995), *HRT-HOOD(TM): A Structured Design Method for Hard Real-Time Ada Systems*, North-Holland, Amsterdam.

Burns, A., Dobbing, B. & Vardanega, T. (2003), Guide for the use of the Ada Ravenscar Profile in high integrity systems, Technical Report YCS-2003-348, University of York. Available at `http://www.cs.york.ac.uk/ftpdir/reports/YCS-2003-348.pdf`.

FAA (2004), *Handbook for Object-Oriented Technology in Aviation (OOTiA)*. Available at `http://www.faa.gov/certification/aircraft/av-info/software/OOT.htm`.

Ghazalie, T. M. & Baker, T. P. (1995), 'Aperiodic servers in a deadline scheduling environment', *Real-Time Systems* **9**(1), 31–67.

IEEE (2003), *1003.13-2003 IEEE Standard for Information Technology - Standardization Application Environment Profile- POSIX Realtime and Embedded Application Support (AEP)*.

ISO (1983), *Reference Manual for the Ada Programming Language*. ANSI/MIL-STD-1815A-1983; ISO/8652-1987.

ISO (1995), *Ada 95 Reference Manual: Language and Standard Libraries. International Standard ANSI/ISO/IEC-8652:1995*. Available from Springer-Verlag, LNCS no. 1246.

ISO (1999), *Ada: Conformity assessment of a language processor*. ISO/IEC 18009:1999.

ISO (2000), *Guidance for the use of the Ada Programming Language in High Integrity Systems*. ISO/IEC TR 15942:2000.

Joseph, M. & Pandya, P. (1986), 'Finding response times in real-time systems', *BCS Computer Journal* **29**(5), 390–395.

Liu, C. & Layland, J. (1973), 'Scheduling algorithms for multiprogramming in a hard-real-time environment', *Journal of the ACM*.

Liu, J. W., Lin, K. J., Shih, W. K., Chuang-Shi, A., Chung, J. Y. & Zhao, W. (1991), 'Algorithms for Scheduling Imprecise Computations', *IEEE Computer* **24(5)**, 58–68.

Motet, G., Marpinard, A. & Geffroy, J. (1996), *Design of Dependable Ada Software*, Prentice Hall.

RTCA (1992), *RTCA/DO-178B: Software Considerations in Airborne Systems and Equipment Certification*, RTCA.

Sprunt, B., Sha, L. & Lehoczky, J. (1989), 'Aperiodic task scheduling for hard real-time systems', *Real-Time Systems*.

Vardanega, T. (1998), Development of On-Board Embedded Real-Time Systems: An Engineering Approach, PhD thesis, TU Delft. Also available as ESA STR-260.

Vardanega, T. & van Katwijk, J. (1999), 'A software process for the construction of predictable on-board embedded real-time systems', *Software Practice and Experience* **29**(3), 1–32.