Ada 2005: A Language for High-Integrity Applications

Dr. Benjamin M. Brosgol
AdaCore

Development of high-integrity software requires a programming language that promotes sound software engineering across domains ranging from small footprint, safety-critical embedded applications to large-scale, real-time systems. The Ada language standard was originally designed in the early 1980s to meet these demands, and a revision in the mid-90s (Ada 95) enhanced its support. More recently, a new version of the language, known as Ada 2005, has introduced additional capabilities based on user experience and advances in language design and software technology. Ada 2005 offers particular innovations that will help make safety certification less costly and improves support for high-integrity systems in three major areas: object-oriented programming, tasking, and real-time features.

A *high-integrity* application is one whose failure would cause unacceptable costs, or for safety-critical systems, create risks to human health or life. Examples of high-integrity applications include aircraft avionics, weapons systems, and shipboard control. Business-critical software can also qualify as high-integrity if a failure could cause significant economic damage, expose confidential data, or have other similar consequences. Developing such applications presents difficult challenges. The programming language chosen has an important effect based on how well it meets the following requirements:

- **Reliability.** The language should support the development of programs that can be demonstrated to work correctly and should help in the early detection of errors in programs. This may sound obvious, but the various goals that a language design seeks to achieve (ease of writing, run-time efficiency, expressive power) can sometimes involve a trade-off with program reliability.
- **Safety.** Although related to the goal of reliability, safety is worth noting as a separate requirement. Informally, safety in a programming language means being able to write programs with high assurance that their execution does not introduce hazards (i.e., the system does not do what it is not supposed to do). This translates into language requirements related to program predictability and analyzability in order to allow the system to be certified against safety standards such as Document Order (DO)-178B [1]. For example, the developer must be able to demonstrate that run-time resources (such as stack space) are not exhausted [2].
- **Expressiveness.** High-integrity applications fall across a variety of domains (real-time, distributed, transaction-oriented, etc.) and the programming language or its associated libraries must provide the appropriate functionality. For example, a real-time system generally comprises a set of concurrent activities (either time or event-triggered) that interact either directly or through shared data structures. It must be possible to express such functionality with assurance that deadlines are met and that shared data are not corrupted by simultaneous access.

This article focuses on how Ada 2005 [3, 4] offers enhancements in each of these areas. The emphasis is on new capabilities, but there is also a brief mention of Ada's existing support for these requirements. Although the reader might not be familiar with Ada, general programming experience with a language such as C, C++, or Java is useful, and some sections assume acquaintance with specialized topics such as object-oriented programming (OOP), multi-threading, and real-time scheduling. An introduction to OOP in an Ada context can be found in John Barnes' Ada 95 Rationale [5]; a comprehensive treatment of concurrency and real-time issues and approaches is provided in [6].

Reliability

Reliability as a language design goal implies support for software engineering. This indicates a prevention of errors with detection at compile time if possible and avoidance of pitfalls where a program does something other than what its syntax suggests. Ada's design was based on these principles. Specific features include strong typing, checks that prevent *buffer overflow*, checks that prevent *dangling references* (i.e., references to data objects that have been reclaimed), a concurrency feature (protected objects) that offers a structured and efficient mechanism for guaranteeing mutually exclusive access to shared data, and an exception handling facility for detecting and responding to deviant run-time conditions such as improper input data. Ada 2005 enhances this support in several areas:

- **OOP.** One of the essential elements of OOP is *inheritance*. A new class (the subclass) is defined as a specialization of a parent class (the superclass), and methods from the superclass are either explicitly overridden or implicitly inherited. However, misspelling a method name when defining a new subclass or adding a new method when revising an existing superclass, may introduce hard-to-detect bugs unless the language provides appropriate features. For example, inheriting from a class that defines a method named *Initialize*, and attempting to override it but misspelling the name as *Initialise,* results in the unintended implicit inheritance of the superclass' *Initialize* method. Dynamic binding to *Initialize* will invoke the superclass' version of the method, which is not what the programmer expected. As another example, adding a method to a superclass when a subclass already has a method with the same name and parameter types causes the subclass's method to override the superclass's method. This, too, causes unexpected effects on dynamic binding. Ada 2005 introduced new syntax that a programmer can use to detect both kinds of errors at compile time. This is more reliable than C++ (which lacks any mechanism) and Java (which provides an annotation that can detect unintended inheritance but does not have a means to detect unintended overriding).
- **Read-only parameters.** Ada's approach to subprogram formal parameters, unlike most other languages, encourages the programmer to think in terms of logical direction of data flow rather than physical implementation (by copy or by reference). Thus, Ada has always supplied the *in* parameter mode, corresponding to data being

passed from the caller to the called subprogram. Assignment to an *in* parameter is prohibited. Even if the implementation passes the actual parameter by reference, the object that is passed cannot be modified through an assignment to the formal parameter. This is extremely useful in ensuring the absence of unwanted updates since the compiler would detect such attempts as errors. However, the Ada 95 mechanism for so-called *access* parameters, which allowed passing a pointer to a declared data object (analogous to * parameters in C) did not include a way to protect the referenced object from being modified. That gap is now filled, with the ability to specify subprogram parameters as pointers to constants. The called subprogram may use the formal parameter to read the value of the referenced object but not to perform assignments to the object. This capability is not new in programming languages (it is available in C and C++, for example, through the * *const* syntax), but it is lacking in Java, which provides no mechanism for constraining a method to have read-only access to the object denoted by a parameter.

- **Assertions.** Ada 2005 has introduced a compiler directive – pragma Assert – through which the programmer can specify a logical condition that is known to be true. In its simplest form, it appears as pragma Assert (*expr*) where *expr* is an expression that returns a Boolean result (i.e., either True or False). When control reaches the point in the program where the pragma appears, *expr* is evaluated. If it is True, then execution continues normally. If it is False, then the Program_Error exception is raised and standard exception handling/propagation semantics apply. This pragma was implemented by several Ada 95 vendors and produced enough general interest to be incorporated into Ada 2005. It provides a convenient and readable notation for specifying many kinds of pre-conditions, post-conditions, and invariants, thus facilitating static analysis and formal reasoning about programs.
- **Avoidance of race conditions during system initialization.** If a program uses concurrency features (known as *tasks* in Ada), there are potential problems at program startup if, for example, a task reads the value of a global variable before the variable has been initialized. Such a hazard is traditionally known as a *race condition*. A new compiler directive in Ada 2005, pragma Partition_Elaboration_Policy, allows the programmer to prevent this problem by deferring task activation until after data initialization.
- **Avoidance of silent task termination.** This hazard in high-integrity systems – the implicit termination of a task because of an unhandled exception or an abort – has been addressed in Ada 2005 with a new mechanism for setting user-defined termination handlers. Such a handler is invoked when the associated task is about to terminate. This allows a controlled response at run-time, for example, keeping track of such events for post-mortem analysis.

Safety

Although DO-178B was originally devised as guidance for commercial aircraft developers, it is applicable more generally on any system where high confidence in correctness is required, and it is being cited increasingly on defense software projects. The document, comprising a set of 66 *guidelines*, focuses on the soundness of the development process and has a particular emphasis on testing as a verification technique. DO-178B, though largely silent about particular languages or language features, implies several requirements that relate to programming language issues:

- **Predictability.** The time and space demands for the system must be predictable. It is unacceptable to miss the deadlines of safety-critical tasks or to exhaust stack space or dynamic memory.
- **Analyzability.** The code must be statically analyzable, both by humans and by software tools. This is needed to support traceability (each software requirement must be traceable to code that meets that requirement, and all code must be traceable back to a requirement that it meets) and structural coverage analysis.

Unfortunately, these requirements conflict with other important goals such as expressiveness and maintainability. The following are examples of conflicts:

- **Dynamic features.** Many modern programming languages, including Ada, have features such as exception handling and concurrency that offer considerable generality, but at the price of run-time libraries that are too complex for safety certification. For compliance with standards such as DO-178B, simple features work best.
- **Object-oriented programming.** The use of OOP in safety-critical systems is a subject that has been attracting considerable attention in recent years and is addressed in detail in the multi-volume handbook, *Object-Oriented Technology in Aviation* [7], evolved under the auspices of the Federal Aviation Administration and National Aeronautics and Space Administration. Two essential characteristics of OOP are *polymorphism* (the ability of a variable to reference objects from different classes at different times) and *dynamic binding* (resolving a method call based on the class of the object that the method is invoked on). But polymorphism implies pointers and thus dynamic memory management, which interferes with predictability. Dynamic binding implies not knowing until run-time the method invoked, which interferes with analyzability.

Ada 2005 addresses these issues in several ways:

- **Language profiles.** With the exception of specialized languages such as SPARK [8], which was specifically designed for safety-critical and security-critical systems, it has always been necessary to define language subsets, or profiles, in order to ensure certifiable run-time libraries and predictable/analyzable application code. The question has been how such subsets have been defined. Ada 95 introduced a compiler directive, pragma Restrictions, which gave this control to the programmer. The programmer can use this pragma to specify exactly which features are needed, thus defining a profile in an *á la carte* fashion. Ada 2005 extends this mechanism with an additional directive, pragma Profile, that allows the formalization of a specific set of features under a common name (this is the way in which the Ravenscar profile, discussed next, has been formalized). In brief, the Ada design recognizes the reality, described in an International Organization for Standardization report [9], that there is no such thing as *the* safety-critical language profile; rather, there are different profiles based on the analysis techniques that are used in certification. For safety-critical systems, Ada 2005 can be regarded as a family of language profiles, with the precise set of features in any given profile defined by the application programmer.
- **Ravenscar profile.** Named for the venue of a workshop where it was first defined, the Ravenscar profile [10, 11] is a set of Ada tasking features that are powerful enough to be used for real-

world applications but simple enough to be certifiable against standards such as DO-178B. A program that adheres to the Ravenscar profile comprises a set of tasks; each task is defined as a loop with a single point where it may be blocked waiting for a timeout or an event. This style straightforwardly expresses a periodic task or a task that handles asynchronous events such as keyboard input or messages from external devices. More general tasking features (such as task abort and asynchronous transfer of control) that would complicate safety certification are prohibited. A major advantage of the Ravenscar profile is that it allows a program to be expressed naturally as a set of tasks, a design that directly reflects the system requirements and is easy to maintain. The traditional alternative is the cyclic executive style, which is brittle in the presence of maintenance changes. Ada 2005 has incorporated the Ravenscar profile into the language, thus making it a formal part of the standard.

- **Safe OOP.** Ada's OOP model offers several benefits in connection with safety certification. First, as noted earlier, Ada 2005 has introduced syntax that helps avoid some subtle OOP errors in connection with inheritance. Second, it is possible, especially through pragma Restrictions, to avoid using OOP features that could interfere with analyzability. For example, the programmer can safely use tagged types (classes), encapsulation, and inheritance, but avoid dynamic binding. Third, Ada 2005 has extended Ada 95's OOP model to include Java-like interfaces, thus making it easier to express multiple inheritance without needing complicated idioms. Finally, automated program transformations are possible that convert a program using dynamic binding to an equivalent version that uses more traditional constructs. If such a tool is qualified (in the DO-178B sense) then the analyzability concerns mentioned here in connection with dynamic binding would be addressed.
- **Safety-oriented pragmas.** Ada 2005 introduces several pragmas that are relevant to safety-critical programs. Pragma Unsuppress can be used to locally enable language-defined checks, thus overriding the effect of a pragma Suppress that may have been applied as an optimization. Pragma Unsuppress is useful in algorithms that depend on the raising of a predefined exception. Pragma No_Return identifies a procedure that never returns to the point of call; it either loops forever (for example, a main routine in a process control system) or else always raises an exception (for example, to indicate detection of some abnormality at run-time). This pragma may be useful for some static analysis tools.

Expressiveness

Many high-integrity applications are real-time systems, requiring language features or libraries that support concurrency and the expression of periodic (time-triggered) as well as aperiodic (event-triggered) activities. A particularly important consideration is the management of *priority inversion*, a situation in which a lower-priority task prevents a higher-priority task from running. Some priority inversions are necessary, such as when a high-priority task needs to be blocked because it is trying to access a shared object that is currently being used by a lower-priority task. The key is to predict the maximum blocking time for each task and to minimize this bound so that deadlines can be guaranteed and available processor capacity can be exploited.

These needs are actually well met by the Ada 95 tasking model, which introduced several important constructs:

- **Standard task dispatching policy.** Ada 95 formalized a traditional fixed-priority scheduler, basically *run until blocked or pre-empted*, with tasks at a given priority level serviced in first-in first-out (FIFO) fashion.
- **Protected objects.** The protected object mechanism captures the notion of an encapsulated object (preventing direct and thus error-prone access to *state* data) with mutual exclusion and condition synchronization. The typical concurrency pattern of multiple tasks interacting through a shared data object (where a task might need to not only acquire a mutually exclusive lock on the object, but also wait until the state of the object satisfies a particular condition) can be expressed clearly, reliably, and efficiently through a protected object and its operations.
- **Ceiling locking policy.** With the ceiling locking policy, a task performing an operation on a protected object will inherit the priority that is defined as that object's ceiling. This policy minimizes priority inversions, and with Ada's semantics for non-blocking protected operations, it prevents certain forms of deadlock.

Ada 95 is especially well suited to off-line (pre-runtime) schedulability analysis, which allows the developer to predict whether all deadlines will be met. Building on Ada 95's foundation, Ada 2005 extends the language's support for real-time systems. The following is a summary of the most prominent new features:

- **Dynamic ceilings.** In Ada 95, the ceiling priority of a protected object must be set at compile time. Ada 2005 is more flexible and allows a ceiling to be changed at run time. This is useful when the ceiling must reflect a changing set of task priorities, for example, due to *mode changes* such as the transitions among the takeoff, cruise, and landing modes for aircraft.
- **Non-pre-emptive scheduling.** In some environments, especially for high-integrity systems, the complexity and overhead of pre-emptive scheduling are not desirable, and the application is prepared to pay the cost of higher latency (less immediate responses to events). Ada 2005 accounts for this need with a new task dispatching policy where a task will run until it either blocks itself (for example, by executing a delay statement) or completes.
- **Round-robin scheduling.** This traditional policy is useful when there is a need for fairness in task scheduling. Ready tasks at the highest priority level are time-sliced at a user-specified interval. This is still a fixed-priority, pre-emptive policy; low-priority tasks are not implicitly bumped in priority based on how long they have been pre-empted. Round-robin scheduling may be summarized as *run until blocked, pre-empted, or time-slice expiration*.
- **Earliest deadline first (EDF) scheduling.** EDF scheduling in Ada 2005 is a dynamic-priority policy in which deadlines and not just priorities are used to dictate which ready task is given the processor. A priority range can be assigned to be governed by the EDF policy. EDF is useful for maximizing system responsiveness, but is less predictable than fixed-priority policies in the presence of overload.
- **Multiple scheduling policies.** Ada 2005 allows different compatible policies to coexist for the same application. This is done by associating task dispatching policies with specific priority levels. For example, the application can reserve a range of low priorities for non-real-time tasks that will

be governed by round-robin scheduling and higher priorities for real-time tasks that require pre-emptive scheduling that is FIFO-within-priorities.

- **Timing events.** Ada 2005 provides a lightweight mechanism for defining asynchronous, time-based events with associated handlers.
- **Group budgets.** Classical scheduling theory deals with aperiodic tasks by grouping them together as a conceptual periodic task with a total budget that is replenished each period; the period is based on the interarrival times of the aperiodic events that the tasks are handling. This functionality can be implemented in Ada 2005 through a *group budgets* package that allows a user-specified handler to run when the budget has been depleted.
- **Execution time monitoring.** Schedulability analysis depends on the correctness of the values provided for task cost (execution time), deadline, and period. This raises the issue of the effect when a task exceeds its cost budget. Ada 2005 addresses this issue through a package that allows tracking of central processing unit time on a per-task basis that also provides user-defined handling of cost-overruns.

In addition to these real-time oriented enhancements, Ada 2005 offers a number of other features that increase the language's expressiveness. It is outside the scope of this article to cover these in-depth but the following are some brief examples of the new features that may be of use in high-integrity applications:

- **More flexible program structuring.** Ada 2005 allows interdependent package specifications, making it easier to model and interface with class libraries as defined in languages such as Java.
- **Unification of concurrency and OOP.** Ada 2005 introduces the concept of a Java-style interface that can be implemented by either a sequential or tasking construct, providing a level of abstraction that is not found in other languages.
- **New libraries.** Ada 2005 adds considerable functionality to the predefined environment. There are new packages, for example, for vectors and matrices, linear algebra, and 32-bit character support. A comprehensive containers library provides facilities somewhat analogous to the C++ Standard Template Library. The definition of high-integrity versions for some of these libraries is in progress.
- **Improved interfacing.** Ada 2005 extends Ada 95's interfacing mechanism, making it easier to construct programs that combine Ada code with modules from C, C++, or Java.

Conclusions

High-integrity software can, in principle, be written in any computer language, but the effort will be simplified by choosing an appropriate language – one that is designed for reliability and safety with expressiveness to capture a broad range of applications including real-time systems. Both the original Ada language and the Ada 95 revision meet these requirements, and Ada 2005 has continued in this vein. Among its enhancements are safer OOP, a certifiable tasking subset (the Ravenscar profile), a way to ensure that pointed-to parameters are read-only, a standard feature for defining language profiles, mechanisms for avoiding hazards such as race conditions at system startup and *silent task termination*, and a variety of new task dispatching policies that are relevant for real-time systems. Importantly, Ada 2005 is real: commercial implementations are in progress, including one that is available at present. Ada 2005 is also at the forefront of real-time study in academia, both influenced by and inspiring research on concurrency, scheduling theory, and related real-time subjects.

Ada has always been an attractive language for high-integrity and safety-critical systems. Advancing the state of the art, Ada 2005 is continuing this tradition and promises to see expanded usage and interest based on its many valuable enhancements.◆

References

1. RTCA SC-167/EUROCAE WG-12. RTCA/DO-178B. Software Considerations in Airborne Systems and Equipment Certification. Dec. 1992.
2. Hainque, Olivier. "Compile-Time Stack Requirements Analysis With GCC." Proc. 2005 GCC Developers' Summit. Ottawa, Canada. 1 June 2005 <www.adacore.com/2005/06/01/compile-time-stack-requirements-analysis-with-gcc>.
3. ISO/IEC JTC1/SC 22/WG 9. Ada Reference Manual – ISO/IEC 8652: 1995(E) With Technical Corrigendum 1 and Amendment 1 (draft 13). Language and Standard Libraries, 2005 <www.adaic.org/standards/ada05.html>.
4. Barnes, John. Ada 2005 Rationale <www.adaic.org/standards/rationale05.html>.
5. Barnes, John. Ada 95 Rationale <www.adaic.org/standards/ada95.html>.
6. Burns, Alan, and Andy Wellings. Real-Time Systems and Programming Languages, Third Edition. Addison-Wesley, 2001.
7. Handbook for Object-Oriented Technology in Aviation. Oct. 2004 <www.faa.gov/aircraft/air_cert/design_approvals/air_software/oot>.
8. Barnes, John G.P. High Integrity Software – The SPARK Approach to Safety and Security. Addison-Wesley, 2003.
9. ISO/IEC JTC1/SC 22/WG 9. ISO/IEC DTR 15942. Guide for the Use of the Ada Programming Language in High Integrity Systems. July 1999.
10. Burns, Alan, Brian Dobbing, and George Romanski. "The Ravenscar Profile for High-Integrity Real-Time Programs." Reliable Software Technologies – Ada Europe '98. Lecture Notes in Computer Science. Number 1411. Uppsala, Sweden: Springer-Verlag, June 1998.
11. Burns, Alan, Brian Dobbing, and Tullio Vardanega. Guide for the Use of the Ada Ravenscar Profile in High-Integrity Systems. YCS-2003-348. York, United Kingdom: University of York, Jan. 2003.

About the Author

Benjamin M. Brosgol, Ph.D., is a senior member of the technical staff of AdaCore. He has been directly involved with the Ada language since its inception as a designer, implementer, user, and educator. Brosgol was awarded a Certificate of Distinguished Service and a Certificate of Appreciation by the Department of Defense, honoring his contributions to the Ada language development. He holds a bachelor of arts in mathematics (with honors) from Amherst College and a doctorate in applied mathematics from Harvard University.

AdaCore
104 Fifth AVE 15th FL
New York, NY 10011
Phone: (212) 620-7300
Fax: (212) 807-0162
E-mail: brosgol@adacore.com