print | close

11 Myths About Ada

Electronic Design
Ben Brosgol
Thu, 2016-03-10 11:27

1. Ada is dead.

"Out of sight, out of mind."

If we're not reading front-page news in the cyber media about some programming language, then obviously it must be dead, right? Well, no. A language can be used successfully, but without great fanfare and without attracting a lot of attention. And failures are much more interesting to read about.

Related

Comparing Ada and C

What's the Difference Between Ada and SPARK?

Ada Offers Advantages Over C And C++

Ada has settled into an important niche—high-reliability real-time systems, especially those with safety and/or security requirements—where it continues to play a major role. A sampling of projects, including financial systems, railway control, air traffic management, satellite software, and commercial and military avionics, among many others, may be found at www.adacore.com/customers/. These aren't simply upgrades of "legacy" Ada projects; many are new programs where Ada was selected based on technical merit and lifecycle cost savings.

So Ada is very much alive and its vital signs continue to be strong. It might not be front-page news, but don't expect to find it in the obituaries.

2. Ada code is slow and bulky.

False and false. The most commonly used Ada compiler is based on the gcc (GNU Compiler Collection) technology, with the same code generator used for Ada and C. So you'll see the same performance (speed, code size) for Ada and C on language constructs that have the same semantics. Okay, but what about all that checking code generated by Ada? For example, to make sure that an index doesn't stray, either innocently or mischievously, beyond the bounds of an array. A couple of responses to that point:

• This sort of check will detect "buffer overrun" bugs. Ada's semantics—raising an exception (which can then be handled with an appropriate response)—has well-defined behavior. This is preferable to the major vulnerability that would be introduced if the program continued execution with an unspecified effect.

• The run-time check can be suppressed after the developer has verified, either through static analysis or sufficient testing, that the check cannot fail.

le size is determined not only by the generated code, but also by the run-time libraries included in the
cutable. And, indeed, Ada’s dynamic allocation, exception handling, and concurrency features all require run-time support. But this issue can be dealt with in several ways.

One is a compiler directive known as pragma Restrictions, introduced in Ada 95. Through this pragma, the programmer can specify features that aren’t used and thus will not include run-time support. And Ada compilers nowadays, especially ones intended for embedded targets with limited code and data space, come with specialized stripped-down run-time libraries corresponding to various subsets (or “profiles”) that are most useful in practice. In addition, an implementation can perform optimizations to remove code that’s never invoked; for example if a package (an Ada module) consists of multiple subprograms but uses only a small number.

3. Ada doesn’t let you get “down and dirty” with the hardware.

With features such as object-oriented programming and generic templates, Ada certainly is a high-level language, and platform independence was a major goal. But Ada can meet that goal and support the lowest of low-level programming, where you may need to deal with “raw bits” and defeat the language’s type checking. Here’s a sampling of the available features:

• An *unchecked conversion facility* that interprets a value (bit pattern) from one type as though it were from some other type

• An *unchecked deallocation facility* that frees dynamically allocated objects

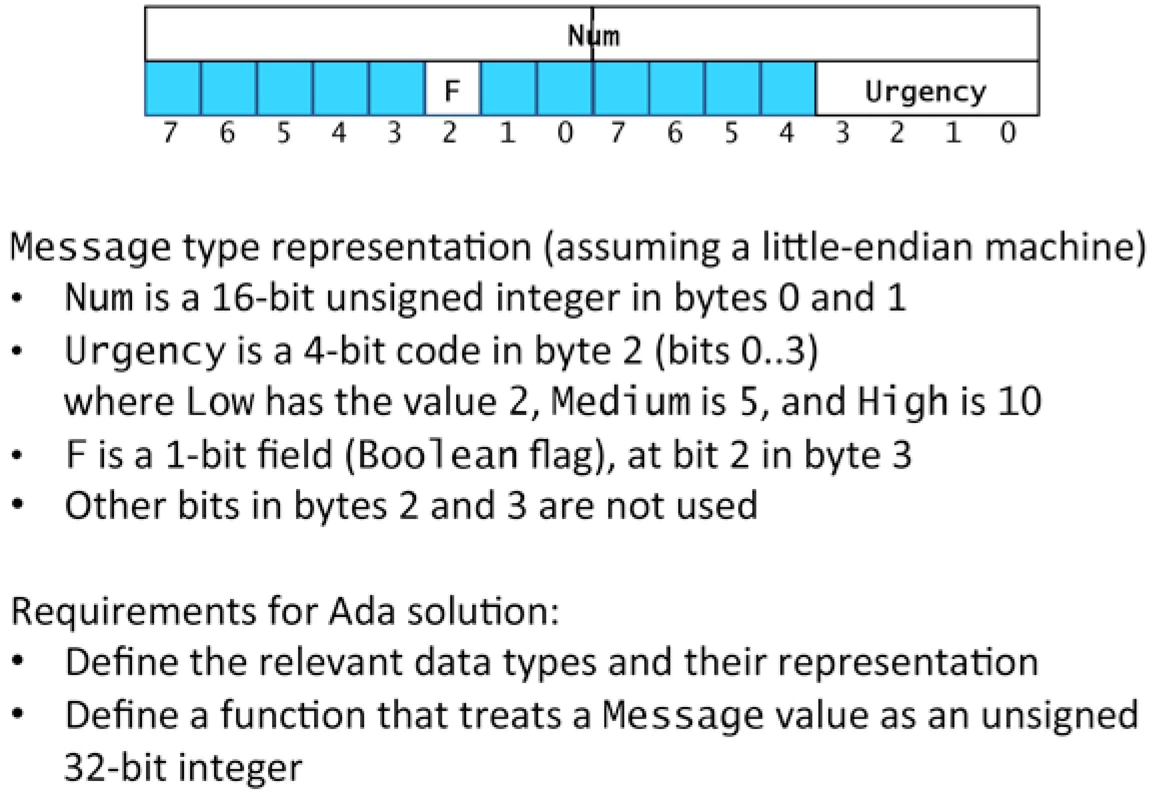
• *Representation clauses* to establish the exact layout (storage unit and bit position offset) for objects of a record (struct) type and define the internal values for the constants from an enumeration type

• *Address clauses* to specify where program entities (generally global data objects) are to be located

• *Interrupt handling support*

• *Endianness support*

An example of low-level programming in Ada is shown in *Figures 1 and 2*.



Message type representation (assuming a little-endian machine)

- Num is a 16-bit unsigned integer in bytes 0 and 1
- Urgency is a 4-bit code in byte 2 (bits 0..3) where Low has the value 2, Medium is 5, and High is 10
- F is a 1-bit field (Boolean flag), at bit 2 in byte 3
- Other bits in bytes 2 and 3 are not used

Requirements for Ada solution:

- Define the relevant data types and their representation
- Define a function that treats a Message value as an unsigned 32-bit integer

4. Ada is stuck in the 1980s.

Anyone who believes this myth has been out of touch with programming language developments for the last few decades. The Ada language underwent a major upgrade in the late 1980s and early 1990s. The resulting Ada 95 version of the standard introduced *bona fide* object-oriented programming (OOP) features (with inheritance, polymorphism, and dynamic binding), as well as a number of other significant enhancements such as the Specialized Needs Annexes for Systems Programming and Real-Time Systems, a structured concurrency feature for state-based mutual exclusion, and more.

The Ada 2005 language included OOP support for Java-like interfaces and a library for container-like data types. And the latest version of the language standard, Ada 2012, brought in contract-based programming (subprogram pre- and post-conditions, type invariants), additional support for multiprocessor and multicore configurations, and other enhancements.

The various revisions to the Ada language, which have been under the auspices of the International Organization for Standardization (ISO), have kept up with developments in both the software and hardware arenas, and in some cases the language has advanced the state of the art. For example, the Ravenscar profile, a subset of concurrency (tasking) features that has a simple implementation with a small footprint, makes it possible to use tasking in an application that requires safety certification. And the contract-based programming features allow for a novel hybrid verification approach that combines formal methods (when the SPARK 2014 subset of Ada 2012 is used) and traditional testing.

5. Ada requires a whole new way of thinking about programming.

Some Ada enthusiasts in the early days of the language (mid to late 1980s) may be responsible for this myth, since portraying Ada as a revolutionary advance was good for publicity. But there were technical as well as marketing reasons why the language wasn't named "Robespierre," "Trotsky," or "Che," and you don't need to earn everything that you know about programming before picking up an Ada textbook. Ada does make it ier to use techniques that were a bit new in the 1980s (such as object-oriented design, where a software architecture reflects the kinds of entities that the system deals with), but it can be used perfectly well for the more traditional procedural style of programming.

6. Ada requires special compiler technology and does not fit in well with other languages.

This myth also dates back to the 1980s, when address space limits on the host machine made it a challenge for Ada compilers to handle large programs. Most compilers of that era used software paging and a centralized compile-time database to deal with separate compilation issues. But progress in compiler technology and exponential advances in computer capacity (address space, speed, storage size) are some of the events that have overtaken this issue. Current compilers for Ada can use the same techniques as for other languages, and those based on gcc can deal comfortably with programs consisting of modules from Ada and from other languages like C, C++, and Fortran.

Ada's standard interfacing features make it straightforward and implementation-independent to compose such mixed-language programs: Ada code can invoke functions or access global data from modules in other languages, and in the other direction it can make global subprograms and data available to foreign-language programs. So, for example, graphical user interfaces can be (and have been) written in Ada, with Ada functions registered as callback routines, even though the underlying graphical software is written in C or some other language. Ada gets along quite nicely with her neighbors these days.

7. Ada is not used in academia.

It's quality, not quantity, that matters, and Ada has seen significant success in both research and teaching over the years. Ada has always attracted the attention of researchers in real-time systems. Some of the seminal work on rate-monotonic scheduling was influenced by Ada's tasking model, and a biennial meeting (the International Real-Time Ada Workshop) brings together the academic and industrial communities to explore how the language can evolve to support new real-time paradigms. Institutions such as the University of York (UK), Cantabria University (Spain), and Polytechnic University of Madrid (Spain) are at the forefront, and the results of their research have made their way into the Ada language standard in the form of task-dispatching policies and related features.

```
with Interfaces;               -- Defines the types Unsigned_16, Unsigned_32
with Ada.Unchecked_Conversion;  -- Generic function
package Message_Pkg is
  type Urgency_Type is (Low, Medium, High);
  for Urgency_Type use (Low => 2, Medium => 5, High => 10); -- Internal representation
  for Urgency_Type'Size use 4; -- 4 bits

  type Message is
    record
      Num     : Interfaces.Unsigned_16;
      Urgency : Urgency_Type;
      F       : Boolean;
    end record;

  for Message use
    record
       Num      at 0 range 0..15; -- Byte 0, bits 0 through 15
                                  --  thus Bytes 0 and 1
       Urgency  at 2 range 0..3;  -- Byte 2, bits 0 through 3
       F        at 3 range 2..2;  -- Byte 3, bit 2
    end record;

  function Message_to_U32 is
     new Ada.Unchecked_Conversion(Source => Message,
                                  Target => Interfaces.Unsigned_32);
  -- Invoke Message_to_U32 on a Message value to obtain the same bit pattern interpreted
  -- as an Unsigned_32 value

end Message_Pkg;
```

On the teaching front, as a modern methodology-neutral language Ada has been used to teach software-engineering principles in general computer-science courses as well as specialized topics in real-time systems or programming-language survey courses. The GNAT Academic Package (GAP), which provides Ada technology at no charge to institutes teaching Ada, currently counts more than 200 colleges and universities from 35 countries as members.

As one example of Ada in an undergraduate setting, students at Vermont Technical College in the U.S. used the SPARK language (a formally analyzable subset of Ada) to develop the software for a CubeSat satellite that recently completed a successful two-year orbital mission. SPARK was chosen because of its reliability benefits. The students had no previous experience in Ada, SPARK, or formal methods, but were able to quickly come up to speed.

Of the twelve CubeSats from academic institutions that were included in the launch, the one from Vermont Tech was the only one that completed its mission. Many of the others met their doom because of software errors. The Vermont Tech group credits its success to the SPARK approach, which, for example, allowed them to formally demonstrate the absence of run-time errors.

8. Ada doesn't have many tools beyond the compiler.

This is another myth that dates back to the days of the initial Ada implementations in the mid-1980s. Early Ada environments were indeed somewhat spartan, for several reasons. First, the language was standardized before implementations were available (this avoided the problem faced by other languages when standardization had to cope with incompatibilities across different compilers), so it's not too surprising that the initial implementations were lacking in amenities.

Second, the policy of the U.S. Department of Defense at the time was to only accept Ada compilers that had been validated against an extensive test suite. This helped to prevent the subset/superset problem that had plagued them with earlier languages. As a result, early implementations were focused primarily on getting the ıpiler to successfully pass the validation test suite, at the expense of other items such as supplemental tools l libraries.

But this is ancient history. Current Ada implementations come with visual integrated development environments and extensive toolsets (debuggers, test case generators, deep static analyzers, source browsers, coverage analyzers, pretty printers, etc.) either bundled with the compiler or available as extras from the compiler vendor or third parties. Ada also has an extensive predefined library together with interfacing facilities that make it straightforward to access libraries from other languages such as C.

9. Ada is too complex.

This is really an amalgam of two sub-myths: Ada is too complex to implement, and Ada is too complex to learn.

The implementation issues were raised very early in the history of the language. In part, this happened because the design, and in fact the requirements that drove the design, were perceived by some to be overly ambitious, with troublesome feature interactions such as exceptions and concurrency. Also, the early-to-mid 1970s saw a backlash against large and complex programming languages like Algol 68 and PL/I, and simple languages such as Pascal were gaining a foothold not only for teaching, but also in industry.

However, experience since then has shown that the pessimism was overly exaggerated. Ada compiler developers were (and are) smart people who know how to deal with language complexity. Where issues existed—for example, the initial tasking design presented optimization challenges—subsequent revisions to the language have offered solutions. In any event, the ensuing Ada compilers used to develop the software in many critical systems serve as counterexamples to the “too complex to implement” claim. “Competing” languages like C++ are at least as complicated, and even C has subtleties in features such as sequence points that belie its perceived simplicity.

The “too complex to learn” claim is also debunked by the evidence. Students needn’t master the entire language, and this author’s experience as an Ada instructor (corroborated by colleagues who also teach Ada both in industry and in academia) is that the language is readily learned.

Challenges exist in teaching Ada because of feature interrelationships. However, by explaining the rationale behind the various design decisions (what problems are encapsulation, object orientation, generic templates, concurrency support, etc., trying to solve) and providing carefully chosen examples, instructors can convey the essence of the language. The training can be online and not just live; for example, the http://university.adacore.com/ site comprises a series of interactive courses on the latest versions of Ada and SPARK.

10. Ada was designed by committee.

This myth must have originated with someone who misinterpreted the lengthy list of acknowledgments in the foreword to the *Ada Reference Manual* and knew nothing about the chief architect of the original design (the late Jean Ichbiah). The foreword does in fact list a large number of individuals who contributed to the language design at various levels, but Dr. Ichbiah was a talented computer scientist who combined technical expertise, a keen sense of language aesthetics, and a forceful personality.

He could sift through the variety of often-conflicting technical proposals on specific points of language design, understand the tradeoffs, and distill and integrate the ideas into a coherent syntactic and semantic structure. This was all the more impressive since a number of the features were new at the time, fresh out of academia or research (generic templates, exception handling, encapsulation, concurrency support). Dr. Ichbiah realized them in Ada, consistent with the overall language architecture.

Most software problems are due to poorly specified requirements rather than coding errors, the programming language doesn't matter that much.

The antecedent in this myth may be true, but the consequent doesn't follow (and indeed the litany of woes from C's notorious "buffer overflow" vulnerability demonstrates that the programming language does matter). The idea that the language choice isn't critical, or that it can be made independent of decisions that affect system quality or lifecycle costs, is too narrow a viewpoint. The programming language makes a difference for several reasons:

• The source code is what gets read, maintained, compiled, and (in some domains) certified.

• Languages differ with respect to their "bug resistance" (i.e., preventing errors from being introduced or detecting them early when they do sneak in).

• Languages can address the "poorly specified requirements" problem in part through features that embed requirements into the source program as assertions that can be checked either statically or at run time.

Ada scores well in all three.

More generally, the programming language is part of the "big picture" for a project, since it can help ensure that consistency (integrity) is preserved across the various system and software development and verification activities. Ada and SPARK have a broad role in the software lifecycle because they address a much larger part of what matters to effective system development than do other programming languages.

In short, a programming language can offer leverage to the developer to gain increased confidence in the system's reliability, safety, and security, while also reducing the effort involved in achieving such confidence. Ada was initially designed to achieve these goals, and subsequent revisions have stayed true to this underlying philosophy.

References:

Ada 2012, http://www.ada2012.org/ and http://www.adaic.org/ada-resources/standards/ada12/

SPARK 2014, http://www.spark-2014.org/

Source URL: http://electronicdesign.com/dev-tools/11-myths-about-ada