Ada: Meeting Tomorrow's Software Challenges Today

A White Paper by AdaCore, February 2019

Executive summary

In a report on the "Do's and Don'ts for Software" [1], the Defense Innovation Board in the U.S. advises "Use modern languages and operating systems…. Treat software development as a continuous activity, adding functionality across its life cycle." The Ada language fully supports these "do's" with state-of-the-art features and a growing ecosystem, and is helping developers of critical cyber systems worldwide meet the most stringent software assurance requirements while reducing life cycle costs. For systems demanding reliable, secure and safe software, Ada continues to be the logical and cost-effective choice.

Introduction

The success of a software project hinges on three main elements:

- Personnel: a cohesive team with the relevant skills;
- Processes: an infrastructure governed by sound and effectively managed procedures for quality assurance, version and configuration control, etc.; and
- Technology: languages and supporting tools for developing and verifying the software components.

All three are critical, but the programming language plays a unique role since it determines the ultimate expression of the software's architecture and functionality. The source code is what lives on -- to be reviewed, analyzed, maintained and perhaps ported -- while staffing may change and processes may evolve. A good language and supporting toolsuite provide leverage that can increase a team's productivity by making it easier to detect errors early in the development process, to design and reuse common components, and to maintain / extend existing code with new functionality. And a good language and its support tools can integrate smoothly into an organization's "software factory" infrastructure.

In this paper we will focus on the role of programming language technology as an enabling factor in a software project. In particular, we will show how the Ada language, with its state-of-the-art design and its emphasis on security, meets the needs of modern critical software projects and helps save costs. We will also show how its ecosystem is evolving and growing,

attracting a new generation of computing professionals. The paper is intended for software developers and managers; it does not require previous knowledge of Ada.

Background

Ada has enjoyed a long and successful history since its inception in the early 1980s, advancing the state of the art in programming language technology while addressing the challenges posed by a software and hardware landscape that has undergone sometimes seismic changes. Designed to meet a set of language requirements for critical embedded systems, Ada has gone through three revision cycles (with a fourth one in progress) under the auspices of ISO, the International Organization for Standardization. Some highlights:

- The initial version of the standard, Ada 83, coherently integrated data abstraction / encapsulation, generic templates, exception handling, and structured concurrency support in the context of an efficient strongly typed language for real-time systems. The design marked a breakthrough in language technology, unifying software engineering advances from the research community in a real-world programming language.
- Based on feedback from user experience with Ada 83, the Ada 95 revision added comprehensive support for Object-Oriented Programming (OOP), an efficient and high-level mechanism for state-based mutual exclusion ("protected objects"), an extensible approach to program library organization ("child packages"), and a number of features that make it easier to combine Ada code with components written in C and other languages.
- The Ada 2005 revision extended the language's support for OOP with Java-like interfaces, including a mechanism that allows implementing an interface for either sequential or concurrent usage. Ada 2005 also standardized the "Ravenscar Profile", a set of concurrency features that are simple enough to be used in applications requiring certification under assurance standards such as DO-178B/C or needing to fit in small-footprint targets, but expressive enough for real-world real-time systems.
- Ada 2012 introduced the key concept of *contract-based programming* with constructs such as pre- and postconditions for code modules, which in effect embed low-level requirements in the source code. Although the underlying concept is not new, Ada's innovative approach allows developers to verify contracts either with run-time checks or through static analysis, and it also supports combining traditional test-based verification with mathematics-based formal methods.
- A new version of the standard, known as Ada 2020, is in the process of being finalized. Its major enhancement is support for fine-grained parallelism to help exploit multicore architectures.

All versions of the Ada language standard, along with supporting rationale documents and detailed analyses of technical issues, are available for download through the Ada Information Clearinghouse website: www.adaic.org/.

Throughout its history, Ada has stayed in sync with the technological advances in the computing industry, both software and hardware, while remaining true to its original design philosophy: make it easier to engineer reliable, scalable, maintainable, portable and efficient software that meets the most exacting requirements for safety and/or security.

Ada and Modern Software Technology

The Defense Innovation Board's "Do's and Don'ts" report [1] explains why modernness matters for a programming language:

> *Modern programming languages and software development environments have been optimized to help eliminate bugs and security vulnerabilities that were often left to programmers to avoid (an almost impossible endeavor). .... Treat software vulnerabilities like perimeter defense vulnerabilities: if there is a hole in your perimeter and people are getting in, you need to patch the hole quickly and effectively.*

Ada and its supporting toolsuites fully embody this "security first" approach along with other important elements of modern language technology.

Detection of vulnerabilities in the Common Weakness Enumeration

The MITRE Corporation's Common Weakness Enumeration (CWE) [2], a categorization of cyber security vulnerabilities into a comprehensive and systematically numbered list, has become a *de facto* reference resource to the software community. The programming language can affect an application's susceptibility to CWE vulnerabilities, and, by virtue of its extensive checks and its "safety first" design philosophy, Ada and its analysis tools can prevent or mitigate many of these and thereby reduce development and verification costs.

Among the vulnerabilities that Ada prevents are unsafe pointer usage (CWE 588), confusion between assignment and comparison (CWE 481 and 482) and improper nul termination for strings (CWE 170). These errors are not possible in an Ada program.

More than three dozen other CWEs are mitigated either through run-time checks or through static analysis tools. AdaCore's CodePeer advanced static analyzer for Ada and SPARK Pro formal methods-based verification tool are two such tools, and both have been recognized as CWE-Compatible in the MITRE Corporation's CWE Compatibility and Effectiveness Program.

The vulnerabilities that are detected by Ada, CodePeer and SPARK Pro include several that are among the CWE's Top 25 Most Dangerous Software Errors, such as buffer overflow (CWE 120) and integer wraparound (CWE 128). These are caught at run-time in Ada, and potential occurrences are also detected statically by both CodePeer and SPARK Pro.

A complete discussion of how Ada and AdaCore tools can prevent or mitigate CWE vulnerabilities may be found in AdaCore's guidance booklet on cyber security [3].

Early error detection

Errors are least expensive to correct when they are detected early in the software life cycle. Ada enforces extensive checks that will catch many kinds of defects before they become part of the executable program, and also detects other errors at run-time (either through compiler-generated checking code or by hardware traps) with programmer control over how they should be handled.

Errors caught at compile time include type mismatches, certain kinds of "dangling references" (where an object goes out of scope while still being referenced by a live pointer), parameter misuses (for example, attempting to assign to an "in" parameter), and many others.

One of the more insidious errors in software development is "version skew": linking a set of object modules into an executable when some module depends on an obsolescent version of another compilation unit. This can occur, for example, if the latter unit is changed but not all dependents are recompiled. The Ada rules detect this error (and prevent linking an executable), without the need for an external "make" utility.

Errors caught at run time include:

- Indexing an array with an out-of-bounds value ("buffer overflow")
- Computing an integer operation whose result exceeds the maximum integer value or is less than the minimum integer value ("integer overflow / wraparound")
- Assigning to a scalar variable a value outside the range of that variable
- Dereferencing a null pointer
- Supplying malformed input as part of an input operation

When the error condition for a run-time error is detected, a corresponding exception is raised which can can then be handled by application code.

Contract-based programming

Pared down to its essentials, software development and verification consist in specifying requirements, mapping them to a design and implementation, and using static analysis and testing to achieve sufficient confidence that the requirements are met. Expressing requirements explicitly in the source code, where they can be verified either statically (with appropriate tool support) or with run-time checks, helps to simplify this process -- thus reducing costs.

Ada has included requirements-oriented features since its inception, for example the ability to declare a scalar variable with a specified range, but Ada 2012 has significantly expanded and generalized this capability in a facility known as contract-based programming.

The code fragment below, a procedure that inserts a string into a fixed-length table where duplicates are prohibited, illustrates some of Ada's contract-based programming functionality.

```
type Table is …  -- Fixed-length table of String values

procedure Insert (T : in out Table; Item : in String)
with Pre  => not Full(T) and not Contains(T, Item),
     Post => Contains(T, Item);
```

The Pre and Post syntax formalizes the requirement that the procedure is to meet. The expressions shown for Pre and Post are the procedure's precondition and postcondition, respectively, and may be regarded as contracts between the Insert procedure and its invoking contexts. The precondition must be met wherever Insert is invoked and may be assumed by the code that implements Insert. Symmetrically, the postcondition must be met by the implementation of Insert and may be assumed by the code that called this procedure. Verifying the contracts thus entails two activities:

- Showing that the precondition is satisfied at each point where Insert is invoked
- Showing that, if the precondition of Insert is met, then the postcondition is also met when the procedure returns

The Pre and Post contracts in effect capture the procedure's low-level requirements. The programmer can control whether these contracts are checked at run-time or are ignored by the compiler, and in the latter case they may be amenable to formal verification, for example through the SPARK Pro toolsuite.

The concept of contract-based programming is not new, but Ada's approach is novel. For example, the ability to check contracts either dynamically or statically facilitates "hybrid verification", where critical modules might be verified via formal methods while others are subject to traditional testing techniques. Unlike other languages, contracts in Ada are part of the standard syntax rather than *ad hoc* specialized annotations.

Adaptability / Maintainability

The Defense Innovation Board's report on the "Do's and Dont's of Software" captures the reality of modern software projects:

> *Treat software development as a continuous activity, adding functionality across its life cycle.*

In other words, expect and plan for change. Programming language technology plays a key role, and Ada is well suited to meet this challenge

Programming-in-the-Large

Programming nowadays means not only specifying algorithmic details but also, and no less importantly, managing the complexity that comes with organizing, modularizing and maintaining code bases that may comprise millions of lines of code. Ada helps in several ways, saving effort both during the initial design and in subsequent maintenance. Here is a sampling:.

- A specific language feature, the package, is the unit of modularization and can enforce encapsulation / low coupling with a clear separation of interface from implementation. A system's software architecture, including inter-module relationships, can be depicted as a collection of Ada packages. Since each package defines a distinct namespace, the same names can be declared in different modules without clashing.
- Ada is a full-fledged Object-Oriented Programming (OOP) language. It supports Java-like inheritance (single inheritance of implementation, multiple inheritance of interfaces), allowing a class hierarchy to be extended in a decentralized manner as requirements evolve.
- Even in the absence of OOP, a package can be extended with new functionality by a so-called "child package", without any effect on existing code.

Reusability and portability

Component reuse has been one of the objectives of programming language design since the dawn of the computer era, and a variety of Ada features help achieve this goal and avoid the cost of "reinventing the wheel". Some examples:

- Ada's separate compilation semantics make it easy to reuse existing modules, with full checking enforced across separate compilation boundaries.
- Ada's generic templates offer a rich syntax for parameterizing modules with types and other program entities; for example, a reusable bounded buffer can be parameterized by element type and then reused ("instantiated") with specific types as required. The checking at an instantiation guarantees that mismatches are detected at compile time, in contrast, for example, with C++ where an error might not show up until run-time.
- Ada has a standard facility for interfacing with other languages -- most notably C, C++ and Fortran. Ada code can invoke functions or access data from foreign modules, and in the other direction, code in other languages can invoke Ada subprograms or access Ada data. This makes it easy to introduce Ada into an existing project that is using another language, or to reuse foreign code in an Ada project. Multi-language interoperability is important in modern large systems, and Ada's interfacing facilities provide a portable and efficient solution.
- Ada has an extensive general-purpose predefined environment (math packages, string handling, etc.), and libraries for other languages are readily available from Ada through binding generators supplied by Ada vendors.

Reuse may entail porting a piece of software to a new target environment, and Ada has a number of features that simplify the job of writing platform-independent code. Here are several:

- A numeric type can be defined in terms of its logical properties, for example an integer range or a floating-point precision, and the compiler will automatically choose an appropriate representation.
- Run-time functionality is expressed as high-level features with specified semantics, for example tasking or exception handling, so that programmers do not have to invoke processor- or operating system-dependent functions.

Real-Time Embedded Systems Support

A modern real-time embedded system is typically a concurrent program comprising periodic and event-driven threads of control that communicate with each other, where the parallelism may be either real or virtual. Application requirements include low-level access to the hardware (for example for interrupt handling), predictable time and space consumption, and efficient run-time performance. These requirements present a formidable challenge to a programming language design; Ada meets this challenge.

Concurrency

Ada has a high-level facility for specifying a concurrent program as a collection of program units ("tasks") that can execute with actual parallelism on a multiprocessor or multicore platform, or with simulated parallelism multiplexed under the control of a task dispatcher on a single processor. Ada's inter-task communication and synchronization mechanism includes protected objects, a structured and efficient mechanism for state-based mutual exclusion. A major extension of the language's concurrency support is in the plans for the upcoming Ada 2020 language revision; it will include features for fine-grained parallelism such as parallel loops, which will be of particular benefit for multicore configurations.

The Ada standard defines task dispatching policies that support state-of-the-art scheduling approaches such as Rate-Monotonic Analysis [4]. The language standard also defines a subset of tasking features known as the Ravenscar Profile [5], which are simple enough to be used in applications that require safety certification or that need to fit in small-footprint configurations, but are expressive enough to program the needed application functionality.

The Ravenscar Profile represented a breakthrough in programming language technology. With Ravenscar, a developer can structure the software architecture robustly as a set of tasks with deterministic semantics rather than as a sequential cyclic executive, simplifying both the verification and the maintenance of the system.

Low-level programming

Ada has many high-level features for general-purpose applications, but it is also a systems programming language: when necessary, the programmer can get "down and dirty" with the hardware. Relevant features include a mechanism for specifying the precise bit layout and/or addresses for data structures, an unchecked conversion facility to explicitly treat an object of one type as though it were of a different type (for example, using a pointer as an integer), and a way to express an interrupt handler in Ada. AdaCore's implementation offers additional low-level support including an innovative technique for solving endianness issues.

Predictability and run-time performance

Ada has an efficient run-time data model. Data objects reside either in "static storage" (global data), on a per-task stack (parameters and local variables) or on the heap (dynamically allocated objects). The implementation only needs to use the heap when the program performs an explicit allocation. Declared objects, even if their size is not known at compile time, are stored on the stack, and a specialized mark-release-style mechanism ("secondary stack") can be used to implement functions returning a value whose size is not known until the point of return. Supplemental tools, such as GNATstack from AdaCore, can compute the maximum stack usage of a program, on a per-task basis.

Garbage collection is not required. Ada provides features for heap reclamation -- both explicit and automatic -- but coding standards for critical software typically forbid allocation after system initialization and also forbid deallocation. These restrictions prevent both storage leakage and dangling references.

The effect is that the application developer can accurately estimate the maximum storage that the program will need. And in the time domain, Rate Monotonic Analysis coupled with worst-case-execution time computation tools such as provided by Rapita makes it possible to achieve time predictability and have confidence that all real-time deadlines are met.

Time predictability does not ensure optimal efficiency (and indeed for hard real-time systems the guarantee of meeting all deadlines may compromise average response time). However, Ada's long and successful track record in critical real-time embedded systems serves as empirical evidence that the language's run-time efficiency meets the performance requirements for that domain. And AdaCore's use of the common GCC back-end technology with an Ada front end means that the code quality for Ada is comparable to that of other languages such as C, for constructs that are similar in the two languages.

Cost Savings from Ada

As illustrated in previous sections of this paper, Ada has a wealth of modern features that can reduce the effort throughout most of the software life cycle, from high-level design (packages) to low-level design (contract-based programming) to development and verification (strong typing, tasking, generics, …) to maintenance (OOP, child packages). The claim of cost savings

from such features can be justified qualitatively. For example, with contract-based programming there is no need to separately formulate a code module's low-level requirements and thus no danger of a mismatch between the requirements' formulation and their expression in the code. In the absence of contract-based programming a mismatch might occur, entailing extra effort to detect and correct this defect.

A quantitative estimation of cost savings is much more difficult, but the results of one such study were published in a 2018 report from VDC Research, "Controlling Costs with Software Language Choice" [6]. This report examined the role of programming language technology in the Total Cost of Ownership (TCO) for IoT and Embedded Engineering projects and concluded that "Ada was one language that stood out in its impact on various project success metrics". Based on a survey of more than 700 respondents, VDC's TCO calculator showed significant development cost savings from Ada relative to other languages (specifically C, Java, C++ and C#) across a range of applications on several platforms. The surveyed systems included communications/networking, aerospace and defense, and automotive, deployed on x86- and ARM-based configurations. One example was a communications/networking system on an x86 processor, written in C. Ada offered the highest possible savings of the four languages, 27% as compared with C. This was more than twice the savings of C++ or C# and five times the savings of Java.

The bottom line from VDC's survey and analysis: "Ada's evolution has made it an increasingly compelling option for engineering organizations, providing both a technically and financially sound solution".

The Ada Ecosystem

To be successful, a programming language needs effective and up-to-date tool support across a range of development platforms, a well-defined process for maintaining the language definition, a vigorous and expanding user community, active interest from academia, and freely available resources such as training materials for new users. Ada meets all these criteria.

Tool support and platform coverage

Ada users are well served by the vendors in the Ada market. AdaCore, the company with the broadest range of Ada products, issues an annual major release of its *GNAT Pro Ada* development environment along with several complementary products: the *CodePeer* advanced static analyzer for Ada, the *SPARK Pro* mathematics-based verification tool for a formally analyzable subset of Ada, and the *QGen* qualifiable and tunable model-based engineering toolsuite. Modern graphical Integrated Development Environments (IDEs) are available for Ada, along with an extensive assortment of static and dynamic analysis tools from compiler vendors and third parties. Ada developers can choose from a wide range of tools including visual debuggers, coverage analyzers, coding standard enforcers, and metrics generators.

Ada environments run on modern host platforms, including x86 Windows, Linux and macOS, for native development. Ada cross compilers target a wide range of 32- and/or 64-bit architectures, such as x86, PowerPC, LEON, and the latest ARM and RISC-V processors. Target configurations include bare metal and a variety of Real-Time Operating Systems: Wind River (VxWorks 6, 7, and 653), Lynx Software Technologies (LynxOS-178), Sysgo (PikeOS), BlackBerry (QNX), Apple (iOS), Android, Embedded Linux, and others.

In critical domains such as airborne systems and transportation, the software needs to be certified against specific assurance standards before being deployed (for example DO-178C for avionics, EN 50128 for railway control and protection). Ada is heavily used for such systems, and a number of certifiable run-time libraries and qualifiable tools are available. Examples include AdaCore's cert and ravenscar-cert libraries, which provide simple but expressive functionality for sequential and concurrent applications, respectively; and tools such as the GNATcheck coding standard enforcer and the GNATcoverage code coverage analyzer.

Important characteristics of modern software technology are openness and extensibility, which allow users to adapt products to meet specialized needs. AdaCore's offerings are Freely Licensed Open Source Software (FLOSS), providing access to the source code, and the GNAT Programming Studio IDE can be tailored to incorporate new tools. For projects with specialized requirements for parsing Ada code, AdaCore's *libadalang* library can be used to develop lightweight static analysis utilities.

International standardization

Users as well as compiler and tool providers need a stable and precisely defined language standard, but one that is open to updates in a controlled and upward-compatible fashion in response to new requirements. Ada meets this criterion. Its evolution is under the auspices of the International Organization for Standardization (ISO), with proposed changes discussed and analyzed by programming language experts worldwide from the vendor and user communities. As noted earlier, the Ada standard is now in its 4th version (Ada 2012), and the next revision is expected in 2020.

Along with the language standard, ISO also maintains the Ada Conformance Assessment Test Suite (ACATS), which comprises several thousand programs that exercise Ada language features. The ACATS tests are updated for each version of the standard and provide a common benchmark for measuring compliance of an Ada compiler with the language standard. The test suite is thus useful both to users and to compiler vendors.

Usage

Here is a sampling of Ada projects by AdaCore customers, including a number of recent ones. Several use the SPARK Ada formal methods technology.

- Airbus A350 XWB - Air Data Inertial Reference Unit (Thales, France)
- Industrial Process Management and Control (ci-tec, Germany)
- 787 Dreamliner - Air Conditioning Control Unit (Hamilton Sundstrand, US)
- Cross Domain Guard for Military Tactical Systems (Rockwell Collins, US)
- C130J Flight Management System (Lockheed Martin, US)
- GIRAFFE Land and Naval Surveillance and Defense Systems (Saab, Sweden)
- Digital Terrain System (UTC Aerospace Systems / Atlantic Inertial Systems Ltd, UK)
- On-board software for Vega C launcher (AVIO, Italy)
- CLARREO Pathfinder Mission (Univ. of Colorado Laboratory for Atmospheric and Space Physics, US)
- International Space Station Communication Subsystem (MDA, Canada)
- iFACTS Air Traffic Management System (NATS, UK)
- Railway Control System (Siemens, Germany)
- Automotive Research Project on Freedom from Interference (DENSO, Japan)
- Multi-Level Security Workstation (Secunet, Germany)
- Total Artificial Heart (Scandinavian Real Heart, Sweden)
- Algorithmic Trading System (Deep Blue Capital, Netherlands)

Details on these and other projects are available at www.adacore.com/industries/.

As the list shows, Ada has spread out beyond the military / aerospace domain and is gaining traction in transportation, automotive, medical equipment and other fields. This usage expansion is due to the increasing need for reliability, security and safety in today's interconnected cyber systems, coupled with a growing recognition that Ada language technology can successfully meet this need.

Ada's spread to new domains is illustrated in AdaCore's participation in the RISC-V Foundation, a non-profit organization chartered to standardize and promote the free and open RISC-V instruction set architecture along with its hardware and ecosystem. In joining the RISC-V Foundation, AdaCore is bringing the Ada and SPARK programming languages to the forefront of the technologies available to RISC-V developers, offering a unique environment for safety- and security-critical applications developed on this platform.

Other indicators of expanding Ada usage include the presence of more than 1300 Ada repositories on github (github.com/) and the free tools and libraries on the Ada Information Clearinghouse site (www.adaic.org/2019/01/free-tools-and-libraries-updates/).

Ada in academia

Ada has been and continues to be an effective language for teaching and research, and the use of Ada in academia is supported and encouraged by an AdaCore-administered initiative known as the GNAT Academic Program (GAP). Under the GAP, accredited academic institutions teaching Ada have access to AdaCore's GNAT toolsuite, including technical support for the instructor. Over 200 members in 35 countries have signed up. The GAP links teachers of Ada

and SPARK the world over and provides a way for members to exchange knowledge and resources.

GAP members have developed impressive software projects including the following:

- Verified Robot Control Software (University of Bristol, UK)
- Muen Separation Kernel (HSR University of Applied Sciences Rapperswil, Switzerland)
- Lunar CubeSat (Vermont Technical College, US)
- Ironsides Secure DNS Server (US Air Force Academy, US)
- Real-Time System Development in Ada using LEGO MINDSTORMS NXT (Polytechnic University of Madrid, Spain)
- ATra Confidential Information Analysis and Protection (Georgetown University, US)

Information about the GAP may be found at www.adacore.com/academia/.

Training materials

Ada is an easy language to learn, from a variety of sources:

- The web site learn.adacore.com/ contains interactive tutorials on the Ada and SPARK languages, including editable and runnable exercises.
- An AdaCore booklet *Ada for the C++ or Java Developer*, available for download from www.adacore.com/resources/, explains Ada concepts by relating them to analogous construct in these other languages.
- Textbooks include John Barnes' *Programming in Ada 2012* and, for SPARK, John McCormick and Peter Chapin's *Building High Integrity Applications with SPARK*.
- The Ada Reference Manual and Rationale can be downloaded or browsed on line at the Ada Information Clearinghouse website: www.adaic.org/ada-resources/standards/.
- Short courses in Ada and SPARK with live instruction, 3 to 5 days in duration, are available from AdaCore. The courses can be conducted at the customer's site and are also scheduled periodically at AdaCore company locations.

Experience from teaching Ada shows that a programmer who knows a language such as C, C++ or Java can quickly become productive in Ada or SPARK. As an example, the software for the Vermont Technical College CubeSat project mentioned earlier was written by an undergraduate student with no previous experience in Ada, SPARK or formal methods. The student came up to speed with little difficulty and produced software (approximately 10K lines of Ada/SPARK code) that worked reliably throughout the CubeSat's entire two-year mission.

GNAT Community Edition

A centerpiece of the Ada ecosystem is the GNAT Community Edition, a free version of AdaCore's GNAT Ada technology. This edition is intended for free software developers, hobby-ists and students and can be downloaded from www.adacore.com/community/. Upgraded annually, the GNAT Community Edition is available on a range of popular platforms for native

development, including x86 Windows, Mac OS X, and GNU Linux. For embedded systems development, cross compilation to ARM ELF, RISC-V and several other target configurations is supported.

Make with Ada competition

Make with Ada is an annual AdaCore-sponsored programming competition with cash prizes. It was conceived as a vehicle to show how Ada or SPARK can be used to program efficient and reliable embedded applications on modern target configurations such as Bare Metal ARM,using the GNAT Community Edition. The competition has attracted entrants from around the world, and winning projects include an Ethernet Traffic Monitor, a Brushless DC Motor Controller, and a MIDI Synthesizer.

One of the goals of *Make with Ada* was to show that embedded system developers with little or no previous Ada experience could easily learn the language and become proficient with its toolset. One of the prize winners in the 2016 competition has confirmed that this goal has been met:

> *I was able to become productive in Ada in about 8 hrs (starting from zero experience) .... Developing the [project] in Ada (a language I don't know well) took roughly the same time as the similar functionality in C and I have more confidence in the Ada code.*

Further information about the competition is available at www.makewithada.org/.

Summing Up

Ada and its supporting tools have evolved over the years to exploit advances in software methodology and hardware design, and they serve as exemplars of modern programming language technology. With a focus on detecting errors early, Ada helps reduce verification effort, and its savings have been borne out in a quantitative study. With a robust and growing ecosystem that includes training resources for new users, the language is spreading into new and critical application areas where reliability, safety and security are needed. Or, from another perspective, forward-thinking developers from critical domains outside Ada's traditional military / aerospace niche are taking note of the language's advantages and adopting Ada for their projects. In the context of the Defense Innovation Board's *Do's and Dont's for Software*, Ada is definitely a "Do".

References

[1] Defense Innovation Board, *Do's and Dont's for Software* (Version 0.6), 2 October 2018. media.defense.gov/2018/Oct/09/2002049593/-1/-1/0/DIB_DOS_DONTS_SOFTWARE_2018.10.05.PDF

[2] The MITRE Corporation. *CWE - Common Weakness Enumeration*. cwe.mitre.org/

[3] R. Chapman and Y. Moy, AdaCore Technologies for Cyber Security. www.adacore.com/books/adacore-tech-for-cyber-security

[4] L. Sha, M. H. Klein, J. B. Goodenough; *Rate Monotonic Analysis for Real-Time Systems*; Technical Report CMU/SEI-91-TR-006; March 1991.

[5] Ravenscar Profile, Section D.13 in *Ada Reference Manual ISO/IEC 8652:2012(E) with Technical Corrigendum 1, Language and Standard Libraries.* www.ada-auth.org/standards/rm12_w_tc1/html/RM-TTL.html

[6] C. Rommel, VDC Research Report. *Controlling Costs with Software Language Choice: How Ada Can Help*. www.adacore.com/vdc-report.pdf