CrossTalk

August 2006 **The Journal of Defense Software Engineering** Vol. 19 No. 8

Ada 2005

4 Ada 2005
This article discusses the creation and key new features of Ada 2005 and compares it to other computer programming languages.
by Richard L. Conn

8 Ada 2005: A Language for High-Integrity Applications
This article examines how Ada 2005 offers particular innovations that will help make safety certification less costly and improves support for high-integrity systems in three major areas.
by Dr. Benjamin M. Brosgol

12 Intuitive Multitasking in Ada 2005
This article describes Ada multitasking as it now exists. This Ada feature builds on the same concepts as Java but is considerably safer.
by Dr. Bo I. Sandén

16 Ada 2005 on .NET and Mobile and Embedded Devices
This article outlines how the A# project seeks to have the best of both worlds, by providing an open-source compilation environment for Ada on the .NET.
by Dr. Martin C. Carlisle

20 The Ada 2005 Language Design Process
This article discusses the Ada language design philosophy, contrasts it with the philosophy behind various other programming languages, and shows how the philosophy helped to ensure a successful, integrated, and consistent result.
by S. Tucker Taft

Software Engineering Technology

22 Maintaining Sanity in a Multilanguage World
This article discusses the reality of technical issues faced by sustainers of legacy code while interfacing Ada, C, and C++ from both a syntactical and runtime perspective.
by Val C. Kartchner

Open Forum

27 Adapting Legacy Systems for DO-178B Certification
This article shows how it is possible to achieve a cost effective approach to enable legacy systems to meet DO-178B certification requirements.
by Paul R. Hicks

ON THE COVER

Cover Design by Kent Bingham

Additional art services provided by Janna Jensen jensendesigns@aol.com

Departments

3 From the Sponsor
From the Publisher

7 Coming Events

15 Call for Articles

19 Letter to the Editor

26 Web Sites

30 Executive Training Seminar

31 BACKTALK

CROSSTALK

76 SMXG Co-Sponsor: *Kevin Stamey*
309 SMXG Co-Sponsor: *Randy Hill*
402 SMXG Co-Sponsor: *Diane Suchan*
DHS Co-Sponsor: *Joe Jarzombek*
NAVAIR Co-Sponsor: *Jeff Schwalb*
Publisher: *Brent Baxter*
Associate Publisher: *Elizabeth Starrett*
Managing Editor: *Kase Johnstun*
Associate Editor: *Chelene Fortier-Lozancich*
Article Coordinator: *Nicole Kentta*
Phone: (801) 775-5555
E-mail: crosstalk.staff@hill.af.mil
CrossTalk Online: www.stsc.hill.af.mil/crosstalk

CROSSTALK, The Journal of Defense Software Engineering is co-sponsored by the U.S. Air Force (USAF), the U.S. Department of Homeland Security (DHS), and the U.S. Navy (USN). USAF co-sponsors: Oklahoma City-Air Logistics Center (ALC) 76 Software Maintenance Group (SMXG), Ogden-ALC 309 SMXG, and Warner Robins-ALC 402 SMXG. DHS co-sponsor: National Cyber Security Division of the Office of Infrastructure Protection. USN co-sponsor: Naval Air Systems Command.

The USAF Software Technology Support Center (STSC) is the publisher of CROSSTALK, providing both editorial oversight and technical review of the journal. CROSSTALK's mission is to encourage the engineering development of software to improve the reliability, sustainability, and responsiveness of our warfighting capability.

S T S C

Subscriptions: Send correspondence concerning subscriptions and changes of address to the following address. You may e-mail us or use the form on p. 15.

517 SMXS/MDEA
6022 Fir AVE
BLDG 1238
Hill AFB, UT 84056-5820

Article Submissions: We welcome articles of interest to the defense software community. Articles must be approved by the CROSSTALK editorial board prior to publication. Please follow the Author Guidelines, available at <www.stsc.hill.af.mil/crosstalk/xtlkguid.pdf>. CROSSTALK does not pay for submissions. Articles published in CROSSTALK remain the property of the authors and may be submitted to other publications.

Reprints: Permission to reprint or post articles must be requested from the author or the copyright holder and coordinated with CROSSTALK.

Trademarks and Endorsements: This Department of Defense (DoD) journal is an authorized publication for members of the DoD. Contents of CROSSTALK are not necessarily the official views of, or endorsed by, the U.S. government, the DoD, or the STSC. All product names referenced in this issue are trademarks of their companies.

Coming Events: Please submit conferences, seminars, symposiums, etc. that are of interest to our readers at least 90 days before registration. Mail or e-mail announcements to us.

CrossTalk Online Services: See <www.stsc.hill.af.mil/crosstalk>, call (801) 777-0857 or e-mail <stsc.webmaster@hill.af.mil>.

Back Issues Available: Please phone or e-mail us to see if back issues are available free of charge.

From the Sponsor

CrossTalk Co-Sponsors Invite Additional Government Organizations On Board

As Ada has continued to grow and evolve to benefit the software community, so has CrossTalk. As part of our charter as the U. S. Air Force's Software Technology Support Center (STSC), we published our first issue of CrossTalk 19 years ago. The Air Force centrally funded the STSC, reflecting its commitment to software process improvement. In addition to creating CrossTalk, the STSC started what is now the Systems and Software Technology Conference (now in its 19th year), and developed a corps of expert software consultants and lead Capability Maturity Model®/Capability Maturity Model Integration appraisers. As a result, the Air Force created an organization that could be used as a resource for improving software development and acquisition throughout the Department of Defense.

The STSC has been very successful for 19 years, but the funding has changed. Unfortunately, faced with extreme fiscal limitations, the Air Force discontinued centrally funding the STSC in 2003. Since that time, the STSC has continued to operate as a primarily fee-for-service operation. To sustain CrossTalk with no advertising revenue and no subscription fee, the publication has been maintained through the combined sponsorship of the Air Force's 76th, 309th, and 402nd Software Maintenance Groups as well as the Naval Air Systems Command and the Department of Homeland Security – all government organizations.

As the current CrossTalk co-sponsors, we invite additional co-sponsors from other government organizations to also become co-sponsors. CrossTalk promotes practices that are customary of world-class organizations, and it makes sense for the best government organizations to align themselves with CrossTalk. If you consider your organization to be in this category, I hope you will consider joining us.

To secure the long-term viability of CrossTalk, we also are investigating commercial support of CrossTalk via commercial advertisements. Our goal is to maintain the same high quality, in-depth technical content and the general look and feel of the existing publication with the addition of an insert listing our commercial supporters. We are interested in your feelings about this. We would also like feedback from our readers in the private sector who may be interested in participating with us as potential CrossTalk supporters. Please provide us your thoughts, inquiries, and support interest and e-mail them to <crosstalk.assocpub@hill.af.mil>.

Randy B. Hill

Randy B. Hill
Ogden Air Logistics Center, Co-Sponsor

From the Publisher

Ada Continues to Evolve

I've watched Ada as it has evolved over the years and have been involved in the controversy as various players debated Ada's future. As a Department of Defense (DoD) contractor, I was sent to Ada courses because the DoD mandated the use of Ada on all future software development, and I had to be ready. I even participated in tutorials explaining Ada 95. But it was not until I became CrossTalk's associate publisher that I was completely immersed in the controversy of Ada's life and death.

Now, as Ada 2005 comes on the scene, many of my colleagues are espousing its benefits, and I am happy to work with them this month to provide information that will enable our readers to make their own decisions regarding Ada's benefits. I have no doubt that the debates over Ada will continue with Ada 2005. My goal with this issue is to provide some of the key information that will allow our readers to make their own decisions.

Elizabeth Starrett

Elizabeth Starrett
Associate Publisher

Ada 2005

Richard L. Conn
Retired (formerly of Microsoft and Lockheed Martin)

Since its formulation in the 1970s, Ada has had a significant impact on the future of government and commercial safety-critical applications as well as the development of other computer programming languages and environments. This article discusses the creation and new key features of Ada 2005, and compares it to other computer programming languages.

The needs of the Department of Defense (DoD) and the capabilities offered by the Ada programming language to support the development of mission-critical, software-intensive systems have matched, by design, for more than two decades. Control of the definition of Ada has passed from the DoD to the American National Standards Institute[1] and then to the International Organization for Standardization (ISO)[2]. Likewise, support for the Ada community has passed from the DoD to a community of users and vendors, thereby eliminating the cost of support for the DoD. The needs of the DoD have changed throughout the years, and the stewards of Ada within ISO have seen to it that Ada changes as well. Operating in an open forum with extensive opportunity for user feedback, the stewards of Ada have caused it to evolve to continue to meet those needs as well as the needs of an international community of users.

Many languages, such as Sun's Java[3], Microsoft's C#[4], and Visual Basic[5], are owned and controlled by companies. Languages such as Ada[6], C[7], and C++[8] are instead defined by ISO with no direct control or enforcement mechanism other than the free market. Sometimes both corporate and ISO have control and ownership in some form, as is the case with Microsoft's C#[9] and Common Language Infrastructure[10]. All languages are impacted by user feedback because of their market ties, but the question of ownership and control can make a decisive difference in language selection within certain application domains. Developers in domains driven by short-term market needs and fluctuations are often happy to follow the latest trends and let others worry about infrastructure support, including computer language support. Developers in domains driven by long-term (sometimes covering many decades) product life cycles may need languages and environments that are more stable in the long run. Its long-term stability and ISO-controlled evolution, as well as its support for safety-critical, high-integrity, large-scale systems development can make Ada attractive to users developing systems with a long life cycle such as the DoD.

Today, Ada is used in many domains in many ways. A theme in several of these domains is that the systems using Ada have long life cycles, are safety-critical, are mission-critical, are large (often containing several million lines of code), and they require high levels of reliability. So it is not surprising that we are also finding Ada in use in several future systems. Ada continues to be with us (no, Ada is *not* dead as some people have come to believe), and it continues to evolve in the form of Ada 2005. ISO's definition for Ada 2005 was *put to bed* in November 2005 and should be officially approved through an international ballot in November 2006. This article presents an overview of Ada 2005 and presents information on how it relates to previous versions of Ada and other computer programming languages.

> *"... the systems using Ada have long life cycles, are safety-critical, are mission-critical, are large (often containing several million lines of code), and they require high levels of reliability."*

Stewards of Ada

Ada has been developed in an international open forum sponsored by ISO and the International Electrotechnical Commission (IEC), specifically ISO's Joint Technical Committee (JTC) 1. JTC1 is assigned to deal with all standardization activities in the domain of information technology. As of March 2006, 28 countries participate in JTC1, with another 42 countries registered as observers; JTC1 is responsible for 538 ISO standards. JTC1 is divided into subcommittees (SCs). SC22 deals with *programming languages, their environments, and system software interfaces*; SC22 is divided into working groups (WGs). WG9 is responsible for the *development of ISO standards for the programming language Ada*. The people participating in ISO/IEC JTC1/SC22/WG9, which include users, vendors, and language lawyers, are the stewards of Ada; James W. Moore of the MITRE Corporation is the convener of WG9. People from the following countries are most actively involved in WG9: Canada, France, Germany, Japan, the Netherlands, Russia, Spain, Switzerland, the United Kingdom, and the United States. For more information on ISO and Ada 2005[11], visit <www.iso.org/iso/en/ISOOnline.frontpage>.

Overall Goals for Ada 2005

There are two overall goals for Ada 2005:

- Enhance Ada's position as a safe, high-performance, flexible, portable, interoperable, concurrent, real-time, and object-oriented programming language.
- Further integrate and enhance the object-oriented capabilities of Ada.

As always, upward compatibility with previous versions of Ada is a prime concern. Note that Ada 2005 will be published as an *amendment* to Ada 95, not a new language.

Over the decades, the developers of programming languages have been learning from each other. Ada has influenced the development of Java, C++, Visual Basic, and even the Microsoft .NET Framework. Likewise, Ada has been influenced by more than 30 other languages, including Java, C, and C++. The stewards of Ada designed Ada 2005 to offer the following:

- At least the safety and portability of

Java.
- At least the efficiency and flexibility of C and C++.
- An open, international standard for real-time and high-integrity system development.

Ada can also take advantage of the enormous libraries of reusable components created by the developers of other languages and computing environments. Dr. Martin Carlisle[12] of the U.S. Air Force Academy, for example, has created the A# compiler, which can readily make use of the Microsoft .NET Framework class libraries and create code in the Microsoft Intermediate Language for compilation by the just-in-time compiler in the Microsoft .NET Framework.

Some Key New Features of Ada 2005

The key new features of Ada 2005 reflect both a *catching up* to update Ada with ideas that have become popular with other programming languages, and a *leaping ahead* to enhance some features of Ada that are more unique to Ada itself.

Safety First

Some people consider Ada to be the premier language for safety-critical software, so the Ada 2005 amendments have been carefully designed to not open any safety holes. In addition, some amendments provide even more safety, in some cases putting even more load on the Ada compiler for catching defects at compile time.

Object.Operation Notation

The familiar and popular *object.operation* notation employed in the most popular languages (such as C++, C#, Java, and Visual Basic) is now available in Ada 2005. Programmers of Ada 2005 may use either the *operation* (object, parameters) mechanism required by Ada 95 or the *object.operation* (parameters).

Extensions to the Open, Predefined Ada 95 Library

Ada 2005 adds the following new standard packages that provide features already found in the implementations and libraries of other languages:
- Environment variables.
- Time access and manipulation, including calendar arithmetic and time zones (including predefined types like DAY_NAME and YEAR_NUMBER).
- File and directory manipulation.
- Containers and sorting (including a predefined generic array sort).
- More string functions and wider characters (type WIDE_WIDE_CHARACTER).
- Linear algebra.

New Features for Real-Time and High-Integrity Systems

As a language to support safety-critical, high-integrity systems, Ada 2005 adds several new standard features that have been informally employed by active Ada software developers in the past, reflecting user interests and needs:
- Earliest deadline first, real-time scheduling.
- Round-robin, real-time scheduling.
- The Ravenscar high-integrity, run-time profile[13].

The Ravenscar profile promotes simple and effective language-level concur-

Safety-Critical Issues and Legacy

A recent report puts the cost of downtime at more than $6 million per hour for financial markets[1]. The report lists this as the extreme value of downtime (with shipping downtime being the low end, at $28,000 per hour). However, in the Department of Defense (DoD), the cost of an hour of downtime could easily be measured not in dollars, but instead measured in human lives. As a result, extreme efforts have to be taken to ensure that our safety-critical software does not suffer downtime. To this end, safety-critical languages have become important, especially in DoD-related software[2]. While this comes as no great shock to programmers that C and C++ are not considered safety-critical languages, Java has matured into a real-time language that is robust enough for real-time safety-critical applications[3]. It is worth noting that many have said that Java closely resembles C++ syntax with Ada semantics[4].

Ada is very much a viable language choice for safety-critical embedded and real-time systems. Some examples follow:
- The C-130J (Hercules) aircraft, manufactured by Lockheed Martin. The C-130J has been adapted for roles such as airlifters, hurricane hunters, tankers, and electronic surveillance.
- The F/A-22 (Raptor) advanced tactical fighter, manufactured by Lockheed Martin in collaboration with Boeing. The stealthy F/A-22, which just recently achieved operational deployment, is the most advanced fighter in the world, used exclusively by the United States.
- The F-16 (Fighting Falcon) fighter, manufactured by Lockheed Martin. The F-16 (the world's first fourth-generation fighter) is the most widely used fighter in the world, with more than 4,000 F-16's deployed by 24 countries in 110 versions.
- The F-35 (Joint Strike) fighter, manufactured by Lockheed Martin. The stealthy F-35 is a multi-role fighter for the next generation, designed for use by the Air Force, Navy, and Marines as well as many allies of the United States.
- The A380 family of aircraft, manufactured by Airbus. The environmentally friendly, fuel-efficient A380 passenger jet airliner is the largest passenger jet in existence, seating as many as 555 people in comfort.
- Many other commercial aircraft. These include the Boeing 777 plus assorted Airbus and Embraer aircraft, as well as future aircraft such as the Boeing 7E7 prototype.
- The National Ignition Facility (NIF) at Lawrence Livermore National Lab. The NIF houses the largest LASER facility in the world, used in experiments in high-energy density and fusion technologies with direct applications to nuclear stockpile stewardship, energy research, science, and astrophysics.
- Air Traffic Control system of the United States. Key elements of the Air Traffic Control system are used in the United States, 60 other countries, the subways of New York City and Paris, unmanned vehicles (ground, aerial, and submersible), and more.

These systems are not only safety-critical systems, but they have a very long lifetime (in terms of decades). As such, legacy issues are raised. These systems must evolve as their requirements change, as their missions change, and as lessons learned on topics develop, helping them become more secure and reliable. It is vital that the software they use also evolve with them, and Ada is positioned to do just that.

Notes

1. See <www.cnsoftware.org/nss2report/Chen-NSS2v.3.pdf>.
2. See "Correctness by Construction: A Manifesto for High-Integrity Software" by Martin Croxford and Roderick Chapman, CROSSTALK, Dec. 2005 at <www stsc.hill.af.mil/crosstalk/2005/12/0512CroxfordChapman.html>.
3. See <www.rtj.org/rtsj-V1.0.pdf>.
4. See "Software Standards: Their Evolution and Current State" by Reed Sorensen, CROSSTALK, Dec. 1999 at <www.stsc.hill.af.mil/crosstalk/1999/12/ sorensen.asp>.

rency, which is essential for safety-critical applications. It is a subset of the Ada 95 tasking model, which contains restrictions to meet real-time community requirements for the following:

- Determinism.
- Schedulability analysis.
- Memory-boundedness.
- Execution efficiency.
- A smaller memory footprint.
- The need to satisfy certification requirements, such as Federal Aviation Administration Aircraft-type certification.

The Ravenscar profile includes, but is not limited to, the following set of restrictions and features:

- No task entries are allowed.
- A maximum of one protected entry is allowed.
- No abort statements are allowed.
- No asynchronous control is allowed.
- No dynamic priorities are allowed.
- No implicit heap allocations are allowed.
- No task allocators are allowed.
- No task hierarchy is allowed.
- The maximum length of an entry queue is one.
- Protected types are not allowed.
- Relative delays are not allowed.
- Requeue and select statements are not allowed.
- Task termination is not allowed.
- User-defined timers and local timing events are not allowed, but execution timers (to help catch task overruns) are allowed (and predefined in the Ada 2005 library).

Popular Interface Approaches

Ada 2005 now supports the notions of interface used in languages such as Java and C#, and architectures such as CORBA. To address a common user need, Ada 2005 adds a new pragma called *Unchecked_Union* for interoperating with C and C++ libraries. Interface types have been added, and Ada 2005 *leaps ahead* with the notion of both active and passive synchronized interface types that efficiently integrate object-oriented programming concepts with real-time programming concepts.

Enhanced Encapsulation

Ada 2005 fully supports both module and object encapsulation. With module encapsulation (as already supported by Java), no synchronization is implied, and access within the module is restricted to private components of a type to the module in which the type is declared (that is, you cannot refer to the internals of a module outside of that module). Private components of multiple objects may be referenced simultaneously. With object encapsulation (as already supported by Eiffel), access to an individual object is synchronized. Only operations inside a protected or task type can manipulate components of a locked object.

> *"Ada has influenced the development of Java, C++, Visual Basic, and even the Microsoft .NET Framework. Likewise, Ada has been influenced by more than 30 other languages, including Java, C, and C++."*

Access Types Enhanced

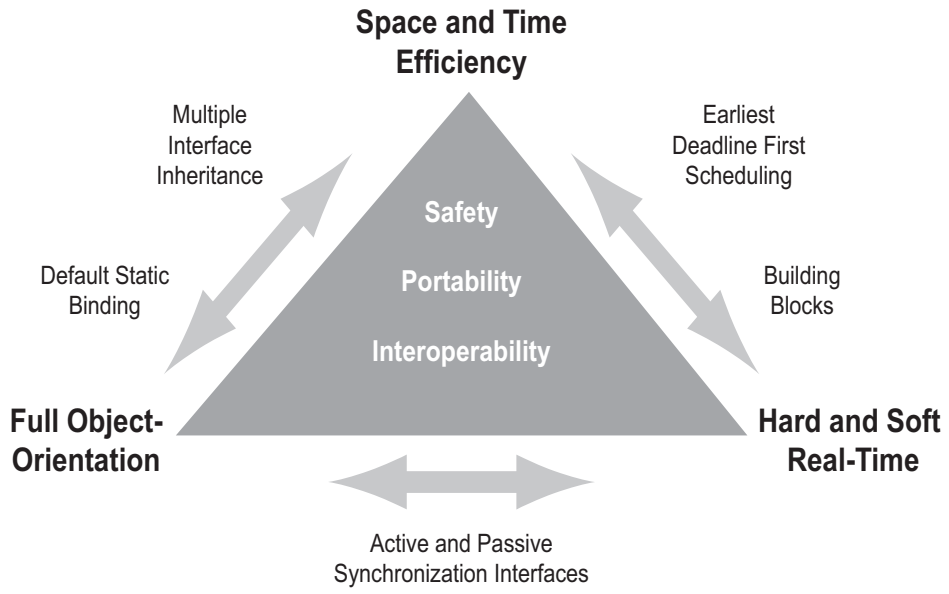
Ada 95 has been considered too rigid by some in its definition of access types. In many cases, a significant number of explicit conversions are required to access anonymous objects and parameters. Ada 2005 adds *anonymous access types* to remove the need for so many conversions.

Dependency Issues Resolved

Addressing a fairly common user need, Ada 2005 adds support for cyclic dependence between types in different packages. *Limited with* clauses allow a limited view of a package, thereby permitting types to be defined across package boundaries.

Figure 1 shows the three key areas addressed by Ada 2005 (full object-orientation, space and time efficiency, and hard and soft real-time requirements), and the key new features related to them such as *earliest deadline first* scheduling.

Figure 1: *Ada 2005: Putting It All Together*



Ada Resources

A number of compilers, development platforms, tools, training and education aids, reusable software components, and other resources are available through the global community of Ada users and vendors. Migration in several forms is now taking place to support development using Ada 2005, and the single best starting point for engaging with the Ada community is the SIGAda Web site at <www.sigada.org>.

The Association for Computing Machinery (ACM) has just completed a review of its special interest groups (SIGs), and while many SIGs are losing membership and viability, SIGAda has been found to continue to be a viable part of the ACM. ISO and SIGAda will continue to be the key focal points for the distribution of information on Ada 2005 and its predecessors and successors.

There are four *major* Ada compiler vendors: AdaCore (GNAT Pro), Aonix (ObjectAda), Green Hills (AdaMulti), and IBM Rational (Apex). There are also several smaller Ada compiler vendors: DDC-I, Irvine Compiler, OCSystems, RR Software, and SofCheck. Finally, there are several vendors providing tools in support of Ada software development includ-

ing, but not limited to: Grammatech, IPL, LDRA, PolySpace, Praxis, and Vector.

Conclusion

Ada 2005 is with us now, and as users become familiar with and ask for the new features, Ada compiler vendors will respond to the users and implement those features. By no means does the evolution of Ada stop now. The needs of the developers of high-reliability, mission-critical, safety-critical, and high-performance systems will continue to change, and by ISO requirement, as long as Ada is an ISO standard, it will continue to be reviewed and updated periodically (on the order of every five years). Ada is here for the long run, and developers of essential systems for the long run should continue to consider Ada.◆

Special Acknowledgement

I wish to acknowledge and give a special thank you to Dr. David Cook for his early review of this article and commentary. Some of his words appear in the sidebar, and his insight and experience, as always, has been appreciated.

Notes

1. See <www.ansi.org>.
2. See <www.iso.org>.
3. Visit the Sun Developer Network's Java Technology Web site at <http://java.sun.com>.
4. Visit the Microsoft Developer Network's C# Developer Center at <http://msdn.microsoft.com/vcsharp>.
5. Visit the Microsoft Developer's Network Visual Basic Developer Center at <http://msdn.microsoft.com/vbasic>.
6. See ISO/IEC Standard ISO/IEC 8652:1995, "Information Technology – Programming Languages – Ada."
7. See ISO/IEC Standard ISO/IEC 9899:1990, "Programming Languages – C" and ISO/IEC Standard ISO/IEC 9899:1999/Cor 1:2001, "Programming Languages – C – Technical Corrigendum 1."
8. See ISO/IEC Standard ISO/IEC 14882:2003, "Programming Languages – C++."
9. See ISO/IEC Standard ISO/IEC 23270:2003, "Information Technology – C# Language Specification."
10. See ISO/IEC Standard ISO/IEC 23271:2003, "Information Technology – Common Language Infrastructure."
11. Ada 95 is formally identified as ISO/IEC 8652:1995, "Information Technology – Programming Languages – Ada." Minor changes to it were approved and published in June 2001 as ISO/IEC 8652:1995: COR.1: 2001: "Technical Corrigendum to Information Technology – Programming Languages – Ada." Ada 2005 will formally be published as an amendment to ISO/IEC 8652:1995.
12. See Martin Carlisle's Web site at <www.martincarlisle.com> for information on and access to his work on the A# compiler, a set of Ada-oriented utilities, AdaGIDE (an Integrated Development Environment for Ada used at the U.S. Air Force Academy), and Rapid Ada Portable Interface Designer, also covered in this issue of CrossTalk.
13. See ISO/IEC TR 24718:2005, "Information Technology – Programming Languages – Guide for the Use of the Ada Ravenscar Profile in High Integrity Systems."

About the Author

Richard L. Conn is retired, having most recently worked for Microsoft, where he was an Academic Relations Manager (a liaison between the research and product teams at Microsoft and many universities in the United States), and Lockheed Martin, where he was a Software Process Engineer for the C-130J aircraft. Conn has more than 25 years of experience in software development and engineering. He has been involved with Ada since 1979, having been involved in the final stages of the Department of Defense (DoD)-1 language competition (DoD-1 later became Ada). He was a member of the Federal Advisory Board on Ada, and received an award from ACM SIGAda for Outstanding Contributions to the Ada Community. Conn's biography is listed in the 2006 edition of *Marquis' Who's Who in America* and the 2006-2007 edition of *Marquis' Who's Who in American Education*. He has a Master of Science degree in computer science from the University of Illinois Champaign/Urbana, and is an Institute of Electrical and Electronics Engineers Computer Society Certified Software Development Professional.

E-mail: rickconn7@msn.com

Coming Events

September 11-15
PSQT '06 North
Practical Software Quality and Testing
Minneapolis, MN
www.psqtconference.com/2006north

September 11-15
RE06
14th IEEE International Requirements Engineering Conference
Minneapolis/St. Paul, MN
www.re06.org

September 18-21
COMPSAC 2006
30th Annual International Computer Software and Applications Conference
Chicago, IL
http://conferences.computer.org/compsac/2006

September 24-27
ICSM 2006
22nd IEEE International Conference on Software Maintenance
Philadelphia, PA
http://icsm2006.cs.drexel.edu

September 25-27
IDGA
Institute for Defense and Government Advancement
FIREPOWER 2006
Silver Spring, MD
www.idga.org/cgi-bin/templates/genevent.html?topic=228&event=10547&

November 12-16
SIGAda 2006
The Annual International Conference on the Ada Programming Language
Albuquerque, NM
www.sigada.org/conf/sigada2006/

2007
2007 Systems and Software Technology Conference
STC Systems & Software Technology Conference
www.sstc-online.org

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

Intuitive Multitasking in Ada 2005

Dr. Bo I. Sandén
Colorado Technical University

As multiprocessors become common, more software must be multithreaded in order to take advantage of the added processing power. Programming in a language with multithreaded support is easier and less error-prone than with stand-alone thread packages. Multitasking in Ada 2005 builds on the same concepts as Java but is considerably safer.

Ada was first released in 1983 with high ambitions. Unfortunately, it came across as large and complicated. Its approach to multitasking, although innovative, proved unsuccessful. It was built on the rendezvous concept [1], which is elegant in theory, but was an unnecessary departure from the long, practical tradition with threads and shared objects as separate, interacting entity types. Ada 95 remedied this problem and further evolved into Ada 2005. Now, the multitasking facilities in Ada are conceptually mainstream, at the same time much less error prone than in other languages.

This article describes Ada multitasking as it now exists. It is intended for developers of multithreaded systems in other languages and for those who may consider redesigning systems to take advantage of multi-core chips and other types of multiprocessors. Programmers in some languages use POSIX threads [2, 3], which are created by routines with a standardized interface. Other routines operate on mutex variables in order to give threads exclusive access to shared data. Support that is built into the language syntax as in Ada and Java is safer and easier on the programmer because it abstracts low-level operations such as the mutex manipulation. A multitask Ada program will run under symmetric multiprocessing.

From the beginning, Ada was designed to meet the predictability and reliability requirements of dependable systems. Over time, those kinds of systems, which are typically embedded real-time systems, formed Ada's niche. Ada supports *hard* real time. In hard real time, computations must finish by certain, absolute deadlines to avoid dire consequences. If an appropriate scheduling algorithm is used, a set of periodic tasks provably meets its deadlines. Ada supports rate-monotonic and earliest deadline first scheduling.

There is an important category of systems that have *no* conflicting, hard deadlines, but still need to be robust. Many of these systems must perform a periodic function reliably. For example, a cruise controller must adjust the throttle with consistent intervals for a smooth ride, but without absolute deadlines. A building monitoring system may need to check every one of hundreds of sensors every two seconds, but there are no dire consequences if the interval is a little longer. Interactive systems need to respond to human input in a consistent and timely manner, but without hard deadlines.

Ada was always intended for high-integrity systems while Java was originally meant for Windows programming and applets. The real-time specification for Java (RTSJ [4, 5]) does not make the language less error-prone[1]. It is easy to make the case that Ada is much safer than Java in this respect [6, 7]. Unlike Java, but like C++, Ada is a hybrid language; you can program entirely without classes. This can be important in hard, real-time environments where some object-oriented features may be deemed too inefficient.

This article focuses on Ada's support for intuitive multitasking without conflicting, absolute deadlines. This intuitive approach can produce elegant, simple, and efficient programs with tasks modeled on entities in the problem domain [8, 9]. A job that is processed in a flexible manufacturing system (FMS) is an example of such an entity [8, 10]. Flexible manufacturing differs from production lines. Each job task waits for access to one workstation at a time and for access to devices such as forklifts. The task describes the life of a job while resources such as the workstations and the devices can be modeled by means of protected objects. Other aspects of the problem also map directly onto Ada features. The following are various language features and examples of how to use them in intuitive multitasking.

Protected Objects

A protected object is a data structure that is encapsulated to make it *task safe*, which means that multiple tasks can operate on it without risk of conflict; each operation on the object always finishes before the next one starts. Protected objects have lock variables, which are inaccessible to the programmer. The compiler inserts the necessary operations on the locks.

A protected type can have *protected operations* of three kinds: *functions*, *procedures*, and *entries*. These are declared in the protected-type specification as in this example:

```
protected type X is
    function F1( ) return Type1;
    procedure P1 ( );
    entry Acquire ( … );
private
    -- Attribute variables
    -- Private operations including
       interrupt handlers
end X;
```

You can also declare a single protected object. The following are differences between protected functions, protected procedures, and entries:

- *Protected functions* are read-only. They cannot change the protected object's attributes and are subject to a read lock; simultaneous function calls on a given object are allowed, but not during a procedure or entry call on the object.
- *Protected procedures* can change the attribute values. They are subject to a write lock; only one procedure (or entry) call at a time is allowed on a given object, and not during any function call.
- Like protected procedures, entries can change attribute-variable values and are subject to the write lock. In addition, each entry has a barrier condition, which must be true for a call on the entry to proceed. For example, an entry Acquire that is to be performed only when the number of items available is greater than zero can be specified in the body of the protected type as follows:

```
entry Acquire ( ... ) when Available > 0 is ...
```

If a task calls *Acquire* when the attribute variable *Available* is equal to zero, it is queued. Each protected object has

one queue *per entry*. This is different from Java, where each synchronized object has a single wait set. Otherwise, a barrier works quite similarly to a wait loop *while (condition) wait( )*, placed at the very beginning of a synchronized operation in Java.

Protected objects are similar to Java's synchronized objects, but much less prone to mistakes and misunderstandings on the part of the programmer [7]. *All* the operations are protected while in Java. In Java, it is up to the programmer to make selected operations *synchronized*.

Use

Protected objects represent shared resources in the problem domain such as workstations and forklifts in the FMS example. A job task acquires a forklift by the call *Forklift.Acquire* on a protected object *Forklift*. It releases the forklift by calling *Forklift.Release*. Release is a protected procedure that increments the variable *Available* and opens the barrier for other job tasks.

While a job is waiting for a resource, its task is queued on a protected entry. This means that *Acquire*'s task queue models the queue of waiting jobs in the factory. Ada gives the programmer provisions for managing such a queue. For example, a task can be removed from the queue if necessary [10].

Requeuing

The Java style with a wait loop as part of the body of a synchronized operation is flexible. For example, a synchronized method in Java can have multiple wait loops; a thread can effectively execute the synchronized operation in segments separated by calls to *wait* where the thread releases its object lock [7]. Each time it is reactivated from the wait set, it enters a new segment with the lock re-established.

The requeue statement in Ada achieves the same effect. It allows a task that is executing an entry to suspend itself and place itself on the queue of the same or another entry. The syntax for requeuing to an entry called *Wait* is as follows:

```
requeue Wait;
```

There are no parameters. The requeuing entry must have the same parameters as the entry in whose body the requeue statement appears.

Use

As mentioned previously, protected objects with operations such as *Acquire* and *Release* can represent resources in the problem domain. Requeuing is the only way to handle a delay during the resource allocation. A high-priority entity may force an entity holding a resource to relinquish it. The high-priority task can requeue itself while the lower-priority task takes appropriate measures to relinquish the resource and call *Release* [8].

Protected Interfaces

Java and CORBA popularized the interface as a syntactic concept [11]. In Ada 2005, a type can implement any number of interfaces in addition to extending a base type. As in Java, an interface is very similar to an abstract class without data where all the operations are abstract[2].

In Ada 2005, protected types can implement *protected interfaces*[3]. Like other interfaces, they allow polymorphism. This addresses, to a degree, an awkwardness in Ada 95, which introduced both protected types and extensible types, but did not combine the two by allowing protected types to be extended [12].

> *"Protected objects are similar to Java's synchronized objects, but much less prone to mistakes and misunderstandings on the part of the programmer."*

If a number of protected types implement the same protected interface *S*, you can reference their instances by means of access variables with a target of type *S'Class*. Ada distinguishes between a type such as *S* and the polymorphic, *class wide* type *S'Class*, which can refer to instances of *S* and its descendants. An access variable is essentially a pointer to an object of the target type.

Use

In a program that deals with devices of different types where each type needs its own driver, you could declare a protected interface with an operation *Initialize* as in the following example:

```
type Device_Handler is protected inteface;
procedure Initialize (D: in out Device_
  Handler) is abstract;
```

The interface is implemented by various device-handler types as *Device_Type1* in this example:

```
protected type Device_Type1 (Device_
  Number : Natural) is
    new Device_Handler with
    procedure Initialize;
          . . .
end Device_Type1;
```

The body of the protected type contains the logic of the procedure *Initialize* for this particular device type. *Device_Number* is a *discriminant*, which allows you to give unique information such as a device number to each instance.

You can now declare an array of pointers to device drivers as follows:

```
Driver_List : array ( … ) of access
Device_Handler'Class;
```

A loop such as the following invokes the initialization procedure appropriate for each array element:

```
for D in Driver_List'Range loop
    Driver_List(D).Initialize;
end loop;
```

For each value of D, the call *Driver_List(D).Initialize* binds at runtime to the *Initialize* procedure appropriate for the type of device at *Driver_List(D)*.

Asynchronous Transfer of Control

Like RTSJ, Ada provides for asynchronous transfer of control (ATC). With ATC, the programmer can arrange for a computation to be cut short if a triggering event should occur before the computation is complete. A common example is an algorithm that iteratively improves an approximate result until it converges to a value with a certain precision. If the calculation does not converge within a certain time limit, a real-time system may need to terminate it and use the best approximation available. The trigger in this case is the event that the time limit is reached.

Ada's ATC syntax is considerably simpler than that in RTSJ [4, 5]. Ada implements ATC by means of abortable tasks. It uses an *asynchronous select* statement such as the following:

```
select delay until Next_Reading;
then abort
    -- "Abortable sequence"
end select;
```

This works as follows: The abortable sequence starts. If it has not ended by the time *Next_Reading*, it is aborted.

You can also express the trigger in terms of elapsed central processing unit time. This is useful in real-time systems with hard deadlines, where each periodic task is given a maximum amount of processor time per period. Finally, the triggering event can be the acceptance of a certain call on a protected entry. The following example illustrates the last case.

Use

In the FMS, each workstation has an input stand, a tool, and an output stand. A job can sit on the output stand of a workstation and wait for its next workstation to become available. The job may still be on the output stand when the next job is finished in the tool and needs the stand. In that situation, the first job must clear the stand and be staged in a storage area; the job task must be prepared to handle whichever comes first of two possible events: the next workstation becomes available, or the job is ordered to clear the stand. Both are entry calls. This is expressed by the following statement:

```
select
   WS.Request;
then abort
         X.Clear;
         -- "Additional statements"
         -- (Stage job in storage)
end select;
```

The trigger, *WS.Request*, is the request for the next workstation, which is accepted when that workstation becomes available. The abortable sequence starts with the call *X.Clear*, which is accepted when the next job is done in the tool. This makes for a race between those two events: *WS.Request* is accepted and *X.Clear* is accepted. One will happen first, and then the other one is aborted. If *X.Clear* is accepted, the call *WS.Request* is aborted and the additional statements execute. If *WS.Request* is accepted, the call *X.Clear* is aborted and the additional statements never execute. The ATC logic arbitrates the outcome of the events even if they happen at practically the same time.

Interrupt Handlers and Timing Events

As a language intended for embedded systems from the beginning, Ada allows the programmer to specify interrupt handlers (RTSJ introduces interrupt handling in Java). In Ada 95 and on, the handlers are protected procedures.

Ada 2005 introduces *timing events* as a means to define code to be executed at a certain time. They are similar to *OneShotTimers*, which are a type of asynchronous events in RTSJ [4, 5]. A timing event can be set to go off either at a certain time or after a certain interval and can be canceled. When the event goes off, it causes a handler to execute. Like interrupt handlers, timing-event handlers are protected procedures.

As with interrupt handlers, the system executes the timing-event handlers; the programmer does not need to supply a task. This simplifies things. In earlier Ada versions, you needed tasks with delay statements to achieve the effect of a timing event.

Use

A car driver can manipulate the driver's side window by means of a lever on the door. Figure 1 is a fragment of a state model of the window. It starts in state *Still*. By pushing the lever down, the driver puts the window in state *Moving_Down*. Releasing the lever takes the window back to *Still*. If the driver holds the lever down for *Time_Amount* milliseconds, the window transitions to the state *Auto_Down*, where it continues down even after the driver releases the lever.

We can define a timing event *Auto_Time* to capture this. It occurs when the window has spent *Time_Amount* milliseconds in *Moving_Down*. It causes the window to enter state *Auto_Down*.

The handlers for the interrupts *Lever_Down* and *Release* and the timing event *Auto_Time* are protected procedures in *Window_Control*, which is a state-machine protected object [8, 9]. It maintains the state of the window in the variable *Wstate* of the enumerated type *State_Type*. *Wstate's* initial value is *Still*. The following are type and instance declarations and part of the specification of *Window_Control*:

```
type State_Type is
   (Still, Moving_Down, Auto_Down, ... .);
Auto_Time : Timing_Event;
Time_Amount : constant Time_Span := ... .
protected Window_Control is
private
   procedure Lever_Down;
   procedure Release;
   procedure Time_Out
         (Event : in out Timing_Event);
   Wstate : State_Type := Still;
end Window_Control;
```

The interrupt and event handlers need not be visible from other parts of the software so they can be declared *private*. I am leaving out statements that tie the interrupt handlers to certain interrupts. The protected body contains the logic of the procedures [8].

In a state machine-protected object such as *Window_Control*, the timing-event handler fits in particularly well among the various interrupt handlers. In this example, where there is no real computation, you need no task at all. A pre-Ada 2005 solution would require a task with a delay statement that calls *Time_Out* when the delay expires.

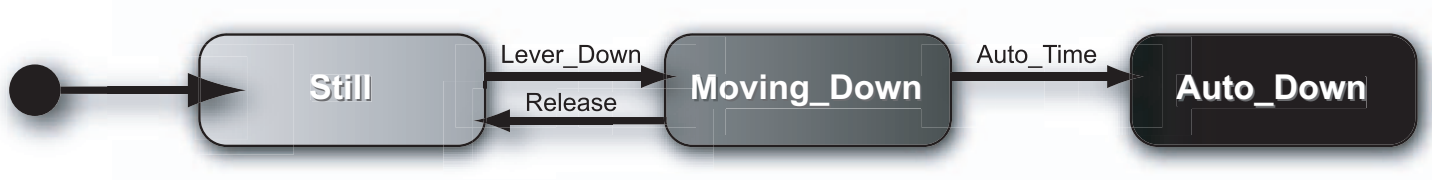
Conclusion

In the mid-80s, soon after Ada 83 first appeared, programmers switched from secure Pascal-like languages such as Ada to insecure C-like languages [13]. Had Ada been an immediate success, things might have gone differently. The original tasking model worked against Ada. That awkwardness is now long gone. Tasking in Ada 2005 is conceptually similar to Java threading, but much safer. Those responsible for critical software development no longer have an excuse to gamble on languages that leave ample room for programmer mistakes.◆

References

1. Burns, A., and A.J. Wellings. Concurrency in Ada. 2nd ed. Cambridge University Press, 1998.
2. Nichols, B., D. Buttlar, and J. Proulx Farrell. Pthreads Programming: A POSIX Standard for Better Multiprocessing. O'Reilly and Associates, 1996.
3. Butenhof, D.R. Programming With POSIX Threads. Addison-Wesley, 1997.
4. Bollella, G., and J. Gosling. "The Real-

Figure 1: *State Diagram Fragment of the Car Window*



Time Specification for Java." IEEE Computer 33:6 (2000): 47-54.

5. Wellings, A.J. Concurrent and Real-Time Programming in Java. John Wiley & Sons, 2004.
6. Nilsen, K. "Applying RAMS Principles to the Development of a Safety-Critical Java Specification." CrossTalk Feb. 2006 <www.stsc.hill.af.mil/crosstalk/2006/02/0602Nilsen. html>.
7. Sandén, B.I. "Coping With Java Threads." IEEE Computer 37:4 (2004): 20-27.
8. Sandén, B.I. Multithreading. Colorado Technical University, 2006 <http://home.earthlink.net/~bosanden/Multithreading>.
9. Sandén, B.I., and J. Zalewski. "Designing State-Based Systems With Entity-Life Modeling." Journal of Systems and Software 79:1 (2006): 69-78.
10. Carter, J.R., and B.I. Sandén. "Practical Uses of Ada 95 Concurrency Features." IEEE Concurrency 6:4 (1998): 47-56.
11. Siegal, J. "OMG Overview: CORBA and the OMA in Enterprise Computing." CACM 41:10 (1998): 37-43.
12. Wellings, A.J., R.W. Johnson, B.I. Sandén, J. Kienzle, T. Wolf, and S. Michell. "Integrating Object-Oriented Programming and Protected Objects in Ada 95." ACM TOPLAS 22:3 (2000): 506-539.
13. Brinch Hansen, P. "Java's Insecure Parallelism." ACM SIGPLAN Notices 34:4 (1999): 38-45.

Notes

1. A safety-critical Java specification is being proposed that carefully defines a small subset of Java and RTSJ to meet stringent reliability, availability, maintainability, and safety requirements [6].
2. Ada interfaces can also have *null* operations, which are concrete but have no effect.
3. *A synchronized interface* can be implemented by protected types and tasks.

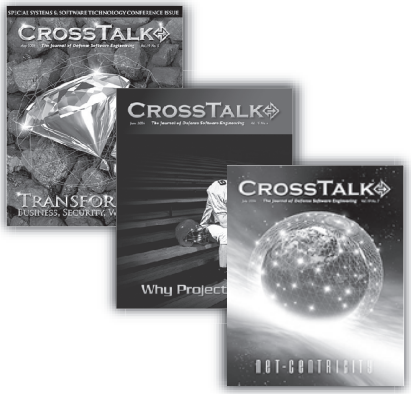
About the Author

Bo I. Sandén, Ph.D., has 15 years experience as a software developer and 20 years teaching sofware engineering. He is now a professor of computer science at Colorado Technical University in Colorado Springs. Sandén's main research interest is language support for multithreading and the design of multithread software. He has a master of science in engineering physics from the Lund Institute of Technology, Sweden, and a doctorate in computer science from the Royal Institute of Technology, Stockholm.

Colorado Technical University
4435 North Chestnut ST
Colorado Springs, CO 80907-3896
Phone: (719) 531-9045
Fax: (719) 598-3740
E-mail: bsanden@acm.org

CALL FOR ARTICLES

If your experience or research has produced information that could be useful to others, CrossTalk can get the word out. We are specifically looking for articles on software-related topics to supplement upcoming theme issues. Below is the submittal schedule for three areas of emphasis we are looking for:



Agile Development
February 2007
Submission Deadline: September 18

COTS Integration
March 2007
Submission Deadline: October 16

Acquisition
April 2007
Submission Deadline: November 15

Please follow the Author Guidelines for CrossTalk, available on the Internet at <www.stsc.hill.af.mil/crosstalk>. We accept article submissions on all software-related topics at any time, along with Letters to the Editor and BackTalk.

CrossTalk
The Journal of Defense Software Engineering

Get Your Free Subscription

Fill out and send us this form.

517 SMXS/MDEA
6022 Fir Ave
Bldg 1238
Hill AFB, UT 84056-5820
Fax: (801) 777-8069 DSN: 777-8069
Phone: (801) 775-5555 DSN: 775-5555

Or request online at www.stsc.hill.af.mil

Name:_____

Rank/Grade:_____

Position/Title:_____

Organization:_____

Address:_____

_____

Base/City:_____

State:_____Zip:_____

Phone:(____)_____

Fax:(____)_____

E-mail:_____

Check Box(es) To Request Back Issues:

Apr2005 ☐ Cost Estimation
May2005 ☐ Capabilities
June2005 ☐ Reality Computing
July2005 ☐ Config. Mgt. and test
Aug2005 ☐ Sys: fieldg. Capabilities
Sept2005 ☐ Top 5 Projects
Oct2005 ☐ Software Security
Nov2005 ☐ Design
Dec2005 ☐ Total Creation of SW
Jan2006 ☐ Communication
Feb2006 ☐ New Twist on Technology
Mar2006 ☐ PSP/TSP
Apr2006 ☐ CMMI
May2006 ☐ Transforming
June2006 ☐ Why Projects Fail
July2006 ☐ Net-Centricity

To Request Back Issues on Topics Not Listed Above, Please Contact <stsc.customerservice@hill.af.mil>.

Ada 2005 on .NET and Mobile and Embedded Devices

Dr. Martin C. Carlisle
U.S. Air Force Academy

Ada is well known for supporting good software engineering practices and for interfacing cleanly with other languages; these features have only gotten better with Ada 2005. The A# project is an open-source implementation of Ada 2005 for Microsoft's .NET Framework. Using A#, programmers can combine Ada code with reusable .NET components, including modules written in C#, as well as legacy component object model components and Win32 Dynamically Linked Libraries. This allows leveraging both the software engineering advantages of Ada and the large amount of reusable libraries written for .NET. Additionally, A# targets portable digital assistants and other mobile and embedded devices.

Similar to the Java Runtime Environment, Microsoft's .NET Framework [1, 2] is attractive to software developers as it provides a large collection of precompiled classes, including security classes that allow dynamic loading of modules. Just as the Java 2 Micro Edition (J2ME) allows Java programs to be run on embedded and mobile devices, the .NET Compact Framework enables .NET applications to be run on these devices. Unlike Java, .NET had a goal of interfacing with legacy Windows code. Therefore, Microsoft provided mechanisms in .NET for easily combining legacy component object model (COM) objects and Win32 Stdcall dynamically linked libraries (DLLs) with new .NET code.

Microsoft developed a flagship language for the .NET Framework: C#. C# is an object-oriented language with a syntax and semantics that are very similar to Java. While C# and Java resolve many of the really bad problems with C and C++ (in particular, buffer overflow vulnerabilities, **=** versus **==**, single character errors, etc.), they still fail to meet many of the almost 30 year old Department of Defense (DoD) Steelman requirements [3] for programming languages that have always been met in Ada. For example, while the latest versions of C# and Java have finally added generics, both languages still fail to provide subtypes to properly model scalar values, and proper enumeration types. C# does have an *enum* type, which at first glance appears to be a proper enumeration type, but does not provide successor or predecessor functions. Additionally, C# and Java still require arrays to be indexed with integers starting at zero (Ada allows the starting index to be specified or allows indexing an array with an enumeration type, such as colors).

Ada, on the other hand, has suffered from a lack of available components (although Ada 2005 does add a new container library). Only a few of the widely used libraries support Ada. Additionally, compiler vendors have been slow to provide compilers for new platforms such as embedded and mobile devices. As a result, despite its engineering superiority, Ada is often not the best tool to get the job done.

The A# project[1] seeks to have the best of both worlds. By providing an open-source compilation environment for Ada on the .NET Framework, A# gives software developers the opportunity to leverage the large amount of reusable .NET classes while also being able to write code in a language that strongly supports good software engineering practices.

Compiling for .NET

One of the key design goals for .NET was supporting multiple different programming languages. Microsoft provides technical support to language developers and has published a list of 27 languages that compile to .NET (not including the four distributed with Visual Studio) [4]. To make it easier for compiler developers to create compilers for the .NET Framework, Microsoft provides the .NET Common Intermediate Language (CIL). The .NET CIL can be viewed as a high-level assembly language, which directly supports object-oriented features such as inheritance, dispatching, and interfaces. Since this intermediate language is object-oriented, compiling an object-oriented language to the .NET CIL is much simpler than compiling to Intel assembly language.

The .NET CIL bears a strong resemblance to Java bytecode. Therefore, JGNAT [5] (Gnu Ada Translator [GNAT] for the Java Virtual Machine) – an open source Ada compiler that compiled from Ada to Java bytecode – was used as a starting point for the compiler. We modified JGNAT to emit CIL instead of Java bytecode. The resulting compiler is called MGNAT (GNAT for Microsoft .NET). Also, we rewrote the Ada standard library packages to call .NET routines instead of those from the Java platform. JGNAT is no longer maintained by Ada Core Technologies, so it does not contain any Ada 2005 features. To support Ada 2005, MGNAT reuses code from the latest GNAT compiler (with some modifications) [6]. GNAT is an open-source Ada 2005 compiler distributed under the Free Software Foundation General Public License. GNAT runs on many different platforms, but not .NET.

To make using the compiler easier, we have modified Ada Graphical Integrated Development Environment (AdaGIDE) [7], a freely available development environment for Ada, so that the A# compiler, MGNAT, can be selected simply by pushing the target button and then selecting the .NET radio button.

Using .NET Classes in Ada

Whenever you mix programming languages, it is necessary to have a language binding that enables communication among components written in different languages. These bindings may be either written by hand or automatically generated. Win32Ada [8] is an example of a handwritten binding to the Microsoft Win32 Application Program Interface (API). The problem with manually writing such a binding is that it is incredibly tedious and quickly becomes stale. As the Microsoft Win32 API developed, Win32Ada was not kept up to date.

A# provides the MSIL2Ada (Microsoft .NET Common Intermediate Language to Ada) tool, which automatically generates bindings. MSIL2Ada is written in a combination of Ada and C# and uses the .NET reflection classes to enumerate all of the classes, methods, and attributes of a .NET DLL, then generates corresponding Ada specifications.

Consider the following C# class:

```
namespace crosstalk {
    public class Class1:Superclass,
    MyInterface {
            public color my_color;
            private string s;
            public Class1(string x) {…}
            public void package() {…}
            public static color get_color() {…}
            private int get_value() {…}
    }
}
```

MSIL2Ada parses the compiled DLL containing this class and generates the following Ada specification (corresponding to only public attributes and methods):

```
with crosstalk.Superclass;
with crosstalk.MyInterface;
with crosstalk.color;
limited with MSSyst.String;
package crosstalk.Class1 is
    type Typ;
    type Ref is access all Typ'Class;
    type Typ is new crosstalk.Superclass.Typ
    and crosstalk.MyInterface.Typ
    with record
            my_color : crosstalk.color.Valuetype;
            pragma Import(MSIL,my_color,
            "my_color");
    end record;
    function new_Class1(This : Ref := null;
            x : access MSSyst.String.Typ)
            return Ref;
    function get_color returncrosstalk.
    color.Valuetype;
    procedure package_k(This : access Typ);
private
    pragma Convention(MSIL,Typ);
    pragma MSIL_Constructor(new_Class1);
    pragma Import(MSIL,get_color,"get_color");
    pragma Import(MSIL,package_k,
    "package");
end crosstalk.Class1;
pragma Import(MSIL,crosstalk.Class1,".ver
1:0:2161:15913","[crosstalk]crosstalk.Class1");
```

In the Ada code above, the compiler directive pragma *Import* specifies that an item is being imported from a different programming language. Several new Ada 2005 features are used in this binding, making it easier to read than similar bindings written in Ada 95. First, Ada 2005 adds interfaces, which are styled after those in Java and C#. In the definition of Class1, the *and* shows how to specify that a tagged type implements an interface. Oddly, in Ada 2005, only child classes are allowed to implement interfaces. This poses no problems in mapping the C# types as all types in C# are derived from System.Object (which does not implement any interfaces).

Second, the new *limited with* clause in combination with new anonymous access types makes it easy to map mutually dependent C# classes. A *limited with* clause is added for each dependent class (as is seen for MSSyst.String).

Reserved words in C# and Ada differ, and C# is case-sensitive while Ada is not, so the *pragma Import* is used to map C# names to Ada names. Note that since *package* is a keyword in Ada, *_k* is added to its Ada identifier. The namespace *System* from C# is renamed to *MSSyst* for similar reasons.

A# was the first Ada compiler to introduce the object.method syntax, which has now become part of the Ada 2005 standard. This means that programmers using both C# and A# can use the same object-oriented syntax for method calls in both languages. In Ada 95, a call to the *package_k* method would have appeared as:

```
crosstalk.Class1.package_k(This=>
Class1_ Ptr);
```

In A# and Ada 2005, this can now be written as:

```
Class1_Ptr.package_k;
```

This is not only shorter, but simpler, as the programmer does not need to worry about what package a class was declared in to call a method on it. This is particularly helpful for non-dispatching methods (those declared with 'Class), as they may be in packages where superclasses were declared.

On the Java Virtual Machine, there were only base types (such as integer and float) and class references. The Microsoft .NET platform allows for the creation of types that stored on the stack are passed by value (hence the name *Valuetype*) instead of by heap reference. To resolve this, we have added the reserved word *Valuetype*. The *get_color* method returns something of type *crosstalk.color.Value-type*. If a type is so named, then the compiler will generate code for it according to the .NET calling conventions for Valuetypes. Since there are no pointers for these types, a full *with* is needed for the *crosstalk.color* package instead of the limited *with* used for other dependencies.

The C# *enum* is another example of a Valuetype. Although, as previously mentioned, it differs from an Ada enumeration type. A# currently maps it to an Ada enumeration. However, since C# *enum* types do not support determining a successor or predecessor, these mapped types cannot use Ada enumeration attributes (such as 'Pos or 'Succ). Unlike Ada enumeration types, .NET enumerations can have multiple names corresponding to the same value. In these cases, a named constant is declared for each additional name. In certain cases, enum values can be combined to create values that have no name (e.g., in the *FontStyle* enumeration, *Bold* and *Italic* are listed – they can be added together to create a Bold Italic style, although this is not listed in the enumeration). When the type allows this, we provide a function (+) for performing such combinations. The C# enum differs so significantly from an Ada enumeration that it would be more accurately mapped to a named integer type (e.g. *type FontStyle is new Integer*). This more precise mapping may be accomplished in a later version.

Another key difference between C# and Ada data types is the use of strings. In C#, strings are stored in 16-bit unicode, while the Ada basic string is eight-bit International Standardization for Organization (ISO) Latin-1. Since it is expected that strings will be commonly passed between the languages, A# provides a unary (+) operator to perform the following conversion.

```
Csharp_String : MSSyst.String.Ref := +
  "hello world";
```

When calling a C# method that takes a string as a parameter, the compiler will automatically insert the conversion:

```
C1 : crosstalk.Class1.Ref :=new_Class1
  (x => "hello world");
```

Extending a .NET class in Ada

A peculiarity that is exposed by the mapping to Ada is the appearance of the *this* parameter in the constructor for Class1. All C# constructors are required, as their first act, to call a parent constructor. Generally, the no-argument constructor of the parent is used; however, this can be changed in C# by using base, as the following:

```
public Class1(string x) : base(x)
```

This indicates that parameter *x* should be passed to the superclass constructor that takes a string as a parameter. When extending a .NET class in Ada, the call to the constructor method is done explicitly in the variable declaration:

```
function New_MenuItem(This : Ref := null)
        return Ref is
    Super : MenuItem.Ref :=
        MenuItem.New_MenuItem
                (MenuItem.Ref(This));
begin
```

```
    return This;
end New_MenuItem;
```

This code seems a bit peculiar, as *Super* appears to be an unused local variable, and the function looks like it will return null; however, the compiler generates the correct code that allocates a new object and calls the parent constructor. The Ada child class also needs to be marked with the convention MSIL (Microsoft .NET CIL) and have its constructor marked as an MSIL constructor (as shown in the package *crosstalk.Class1*).

Calling A# Code From C#

Although a free graphical user interface (GUI) builder tool, Rapid [9], can be used to develop user interfaces for the .NET framework in Ada, Visual Studio [10] provides a much more extensive GUI builder. Consequently, it can be advantageous to use Visual Studio to develop the user interface and then write the rest of the code in Ada.

In this case, you create a DLL from the Ada code instead of an executable. A# comes with mgnatmake, a tool which automatically detects dependencies between Ada source files, performs the required compilations, and combines the results. By default, mgnatmake generates executables, but it can be instructed to generate DLLs instead:

```
mgnatmake msil2ada -o msil2ada_output.
  dll- z -largs /DLL
```

These arguments tell mgnatmake to combine all of the dependencies of the project MSIL2Ada into an output file named msil2ada_output.dll (-o), with no main program (-z) and to create it as a DLL instead of an executable (-largs/DLL).

In Visual Studio, you can simply add a reference to this DLL, and the Intellisense will automatically suggest the Ada methods (Visual Studio automatically creates the appropriate language binding). Because .NET does not allow a namespace and a class to have the same name, it was necessary to add *_pkg* to the end of the final Ada package name, so *P1.P2.P3.Put* would be referenced as *p1.p2.p3_pkg.put*. Also, it is necessary to call the initialization routine generated by the binder explicitly from the C# code as:

```
ada_msil2ada_output_pkg.adainit();
```

The binder output has an additional *ada_* prefix on the package name (which has been given the *_pkg* suffix as previously described).

Uses of A#, Embedded and Mobile Devices, and More Interoperability

The most widely used program implemented using A# is RAPTOR [11] (Rapid Algorithmic Prototyping Tool for Ordered Reasoning). RAPTOR is an open-source visual programming environment designed for use in an introductory computer science class. RAPTOR's visual programming model is based on flowcharting. RAPTOR is being used in an increasing number of universities across the United States and Canada with inquiries from as far away as Japan.

RAPTOR is an interesting use of the A# technology, as it incorporates C# and A# code, as well as a legacy C++ graphics library. Figure 1 shows the interrelation between the various software components in RAPTOR.

The C#, C++, and Ada code are written by hand; the interoperability DLL is generated automatically by Visual Studio when a reference to the COM object is added to the project.

A# is also being used on some defense projects, in particular to port Ada applications to embedded and mobile devices using the .NET Compact Framework. It is a trivial matter to target an embedded or mobile device using A#. Simply adding the-*compact* flag to both MSIL2Ada and MGNAT instructs these tools to generate code suitable for use with the .NET Compact Framework. Two of the platforms available with the compact framework are the Pocket PC and the Smartphone 2003.

A final, recently added piece of interoperability is the ability to interface with legacy Win32 DLLs. In C#, you would add the following code to import from a Win32 DLL:

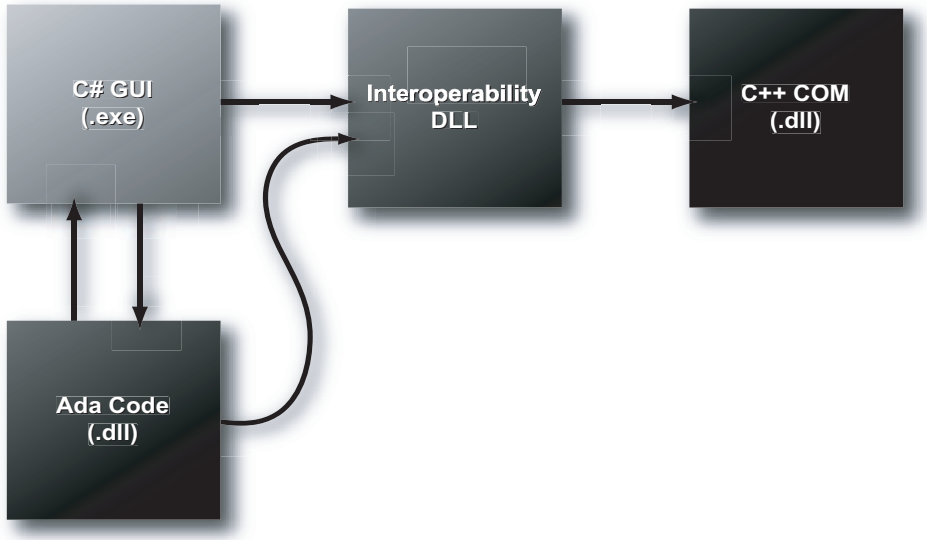
```
[DllImport("adagraph2001.dll")]
public static extern int
    CreateGraphWindow(int size);
```

In A#, you instead do the following:

```
function Open_Graph_Window(Size : in
    Integer)
    return Integer;
pragma Export(Stdcall,Open_Graph_Window,
    "[adagraph2001.dll]CreateGraph
    Window");
```

This also provides a function body (which will be ignored). While it is irregular to use a *pragma Export* to import from another file, this is done because it is necessary to generate code in addition to merely a call, and it was simpler to generate a body on a function marked for export.

Figure 1: *Interoperability Demonstrated in RAPTOR*



Conclusions and Future Work

A# has demonstrated the viability and usefulness of combining Ada with other .NET languages, as well as providing interfacing to legacy Win32 and COM libraries. This allows developers to gain the advantages of Ada's strong typing while also leveraging the vast number of libraries available with .NET and the GUI builder from Visual Studio. Furthermore, A# provides an easy mechanism for getting Ada code running on embedded and mobile devices via the .NET compact framework (e.g. Windows CE, Pocket PC, Smartphone 2003).

There are significant areas for future work. First, we are currently working to fully integrate Ada into Visual Studio 2005. Visual Studio 2005 now allows extensions to be written in .NET languages (Visual

Studio 2003 required the extension to be written using C++ COM objects), so the integration code is also being written in a combination of A# and C#.

Second, version 2 of the .NET framework adds generics. Using the generics built into the framework to implement Ada generics might reduce code size and make the genericity of Ada constructs visible to other .NET languages. This will be a non-trivial effort as the generic model in .NET is not as fully featured as that in Ada (it allows only types as generic parameters). Also, MSIL2Ada and MGNAT need to be updated to allow Ada programmers to use generic classes written in C#.

Finally, while A# has been maintained as an academic project (even though it is in use by defense contractors), it would be preferable to perform technology transfer to the private sector, which has greater resources to develop and maintain this product.◆

References

1. "Introduction to the .NET Framework." DevHood. 7 Mar. 2006. <www.devhood.com/training_modules/dist-a/Intro.NET/?module_id=1>.
2. "Technology Overview: What Is .NET?" Microsoft. 7 Mar. 2006 <http://msdn.microsoft.com/netframework/technologyinfo/overview/default.aspx>.
3. U.S. Department of Defense. Requirements for High Order Computer Programming Languages. "STEELMAN." 1978. 29 Nov. 2005 <www.adahome.com/History/Steel man/intro.htm>.
4. "About Languages: Welcome to the .NET Language Developers Group." Gotdotnet.com. Oct. 2004 <www.gotdotnet.com/team/lang/> Microsoft. (29 Nov. 2005).
5. Comar, Cyrille, Gary Dismukes, and Franco Gasperoni. "Targeting GNAT to the Java Virtual Machine." Proc. of the Tri-Ada 97 Conference. St Louis, Mo., 9 Nov. 1997.
6. "The Libre Site for Free Software Developers." Ada Core Technologies. 29 Nov. 2005 <http://libre.adacore. com>.
7. Carlisle, Martin. "AdaGIDE Home Page." 29 Nov. 2005 <http://adagide.martincarlisle.com>.
8. Taft, S. Tucker. "Win32Ada." 22 June 1999 Averstar and Labtek (29 Nov. 2005) <http://archive.adaic.com/tools/bindings/win32ada/win32 ada.html>.
9. Carlisle, Martin. "RAPID Home Page." 29 Nov. 2005 <http://rapid.martincarlisle.com>.
10. "Microsoft Visual Studio Developer Center." Microsoft. 29 Nov. 2005 <http://msdn.microsoft.com/vstudio/>.
11. Carlisle, Martin. "RAPTOR Home Page." 1 Dec. 2005 <http://raptor.martincarlisle.com>.

Note

1. See <http://asharp.martincarlisle.com> for more information on the A# project. Developers can download A# for free.

About the Author

Martin C. Carlisle, Ph.D., is a professor of computer science at the U.S. Air Force Academy in Colorado Springs, Co. His research interests include programming languages, computer security, and computer science education. Carlisle is the primary author of several open-source software products used worldwide, including AdaGIDE, RAPTOR, and A#. He has a Bachelor of Science in mathematics and computer science from the University of Delaware and a Master of Arts and Doctorate in computer science from Princeton University.

Department of Computer Science
2354 Fairchild DR
STE 6G101
U.S. Air Force Academy
Colorado Springs, CO 80840-6234
Phone: (719) 333-3590
Fax: (719) 333-3338
E-mail: carlislem@acm.org

Letter to the Editor

Dear CrossTalk Editor,

Kevin Stamey's sponsor note, "Why Do Projects Fail?" in CrossTalk's June 2006 issue was encouraging. Software people are finally starting to realize that systems engineering is necessary to their success. What Stamey observes is mostly correct. But he does omit several items, some of which were touched on by the articles in the June issue.

He omits Configuration Management (CM). Without it you are doomed to fail. Who hasn't been burned by some cowboy coder who decided to make an *improvement* without telling anyone, let alone obtaining authorization, delaying testing and causing previously working code modules to fail unexpectedly. Even finding the latest version of a document challenges most organizations.

But CM is really a subset of communication and coordination. When I worked in acquisition, I included a glossary of every term used so there would be no mix-ups, as in Alan Jost's article. Anyone who does not define their terminology is asking for protests, screw-ups, and lawsuits. Why including a glossary isn't standard practice is a mystery. It should continue into the development work by instantiating a project glossary that goes to the level of detail of the units used in calculations.

As Capers Jones alludes to in his article, lack of adequate resources is a root cause of failure. Lack of ethics and moral courage on the part of management and engineering exacerbates the problem, as does outside influences such as political pressure and executives who want to *make the numbers* to get their bonus; congress may cancel funding if progress is not shown. With such a situation, misleading status reports are sure to result, making the situation even more critical later on.

Tim Perkins has the best high-level diagram that I have seen. I infer that it puts too much faith in CMMI-type answers, but it captures the paths to the real root causes. However, Item 150 is a constraint that must be considered in the Systems Architecture; it is not a valid cause of project failure.

Between large, complex, unprecedented systems and small routine, incremental improvements to COTS, there is a wide range of processes that should be used. Processes must be tailored to fit the situation. This requires that competent people be used. Ones who understand, not merely check off boxes on some list. They must truly understand the essence of what they are doing and not just chant the black magic incantations they were promulgated by some professor.

William Adams, PE, Ph.D.
<williamadams@ieee.org>

The Ada 2005 Language Design Process

S. Tucker Taft
SofCheck, Inc.

The Ada 2005 language design process was significantly different from that used by Ada 83 and Ada 95 in that it was based on an essentially volunteer committee rather than a full-time design team. Design-by-committee has a well-deserved reputation as being a sure way to create an awkward and unpleasant collection of disjointed compromises. Nevertheless, the result of the Ada 2005 process produced a language that is even better integrated and consistent than its predecessors. Producing this result depended on having a strong, shared, language design philosophy driving the design decisions. This article discusses this language design philosophy, contrasts it with the philosophy behind various other programming languages, and shows how the philosophy helped to ensure a successful, integrated, and consistent result.

Ada is now entering its third standard incarnation, currently known as Ada 2005. Its earlier incarnations were Ada 83, which was designed in the late 1970s and early 1980s by a team led by Jean Ichbiah, and Ada 95, which was designed in the early 1990s by a team led by this author. In contrast to the earlier incarnations, Ada 2005 was designed by a largely volunteer committee, led by Pascal Leroy. The only member of the committee actually on the *payroll* was Randy Brukardt, who was supported by AdaEurope and the Ada Resource Association in his official role as editor of the new standard.

The lack of a full-time design team to drive and shape the design process created misgivings among some members of the committee who felt it could impede the process. The design-by-committee process has a well-deserved reputation for producing awkward and unpleasant collections of disjointed compromises. The question was whether the Ada 2005 process could sidestep these pitfalls.

With a full-time design team and a clear team leader, the Ada 95 revision benefited from a cohesive vision that kept the design from becoming a scattered combination of ideas. With a volunteer committee, there was a danger that the need to create consensus without the hierarchy present in a design team would result in inconsistencies, that each committee member would be mollified by being given their own *pet* feature, and the language would descend into a balkanized conglomerate of sublanguages.

Ada 2005 seems to have escaped the notorious design-by-committee problems. The proposed changes seem to have brought the language into a state where it is, if anything, more integrated and more consistent. How was this accomplished? In retrospect, the key factor in achieving this desired goal has been a strong, shared, language-design philosophy driving design decisions. This kind of shared philosophy might not have been possible in earlier Ada standardization activities, as the language and the community of users were still relatively new. For the Ada 2005 process, we had a set of committee members with many years of experience both as users and implementors of Ada and a shared vision of what makes Ada powerful and productive, namely its unique combination of safety, flexibility, efficiency, and its real-time support. Our goal in Ada 2005 was to preserve and enhance these strengths while reducing any impediments to productive use.

> *"In retrospect, the key factor in achieving this desired goal [designing Ada 2005] has been a shared language-design philosophy driving design decisions."*

The two *anchors* in the shared vision were safety and efficiency, with safety given more weight – though never absolute precedence – when there was a conflict. Ada's focus on safety is in strong contrast with certain other languages, where the attitude might be expressed as *give programmers very sharp tools and then get out of their way*, although this latter attitude sounds great for *real programmers*. In fact, even the best programmers make mistakes. Part of the Ada philosophy is that *by appropriate human engineering, you can produce a language that is in the end more productive*. The design of the language allows the compiler and the run-time to catch typical programmer errors before they become tedious debugging problems.

To illustrate how this shared philosophy interacted with the Ada 2005 design process, it is useful to study the evolution of a particular Ada 2005 feature as it moved from a perceived language problem – through the debates over the multiple ways to solve it and finally to the ultimate consensus around one particular solution.

Solving the Mutual Dependence Problem

One of the first challenges for the Ada 2005 revision was allowing for mutual dependence among types that were *not* all declared in the same package. As an example, one might have a type representing employees and another representing departments where a department record would include a pointer to the employee who is the manager, and the employee record would include a pointer to their department. In Ada 83 and Ada 95, by using an incomplete type declaration, a software engineer was able to define such mutually dependent types, but only if they were all in the same package. This limitation led to large, unwieldy packages, particularly in the context of object-oriented programming.

Although this mutual dependence problem was one of the first identified, it was one of the last problems solved during the revision process. The problem proved extraordinarily difficult to solve in a way that satisfied the various criteria inherent in our shared design philosophy. In the end, seven different approaches were considered, six of which were considered viable:

1. A new kind of incomplete type called a *separate incomplete type* whose completion is given in a separate library unit, rather than in the same library unit that contains the incomplete type declaration.
2. A variant of the *separate incomplete type* called a *type stub* where the type stub identifies the particular library unit where its completion will be found.
3. A new kind of incomplete type declaration that specifies that the completion will occur in a particular child or nested package of the package containing the incomplete type declaration.
4. A new kind of *with* clause called a *with type* clause to specify that a particular type will exist in a separate package, without requiring the package itself to be compiled prior to the referencing unit.
5. A new kind of compilation unit called a *package abstract* to contain incomplete type declarations that are to be usable (via a *with abstract* clause) as part of a mutual dependence that crosses package boundaries.
6. A new kind of *with* clause called *a limited with* clause that gives visibility on an implicitly created *limited view* of a second package, where the limited view contains incomplete versions of the (non-incomplete) types declared in the second package; *limited with* clauses are allowed to create circular dependencies between packages.

Although each of these proposals had its particular merits, only one ultimately emerged as the best when judged against all of the design criteria. One of the most important criteria was that the feature should preserve the ability to identify all of the inter-compilation unit dependencies by only looking at the name and the *context clause* of a compilation unit (the *context clause* is the set of *with* and *use* clauses that immediately precede a compilation unit). The *separate type* and *type stub* proposals lacked this. Alternative three, allowing a type to be completed in a child, also lacked an indication in the context clause that a dependence on the child existed, though it was given some credit for keeping the unannounced dependence to a unit in the same package hierarchy.

After eliminating the proposals that introduced unannounced dependencies, we were left with the *with type, package abstract,* and *limited with* proposals. The *with type* proposal was abandoned because it did not really solve the whole problem since it did not provide any visibility on an access type, and a mutual dependence between types necessarily involves an access type. Furthermore, it created a namespace with holes in it, where visibility was granted by a *with type P.T*, on a single declaration within package P, without providing visibility on the rest of the visible part of P. This was unprecedented and was inconsistent with a choice made in Ada 95 when designing *with* clauses that mention child units where the entire parent package visible part was included rather than just the named child. An important criteria in our design philosophy has been to try to make consistent choices so that the programmer's intuition about how the language works is reinforced as they learn more of the language rather than being forced to learn new rules in each corner of the language.

The *package abstract* proposal was abandoned primarily based on the criteria of simplicity of use and implementation. Adding a new kind of compilation unit, a package abstract, would be a significant disruption to all existing Ada tools. Forcing the user to decide which types of the package to include as incomplete types in the package abstract felt somewhat arbitrary, and the decision might change repeatedly as the system grew. Although this proposal was abandoned, its heritage can be seen in the simpler-to-use *limited with* proposal. Here, the implementation creates the equivalent of the package abstract implicitly, creating incomplete type definitions for *all* of the types in the original package, while relieving the user of having to perform the potentially error-prone copy-and-paste process manually. Furthermore, implementability concerns were lower because many non-compiler tools could largely ignore these implicitly created limited views.

In the end, there was agreement that the *limited with* proposal was clearly superior to the other five alternatives. But the process of reaching this point was long and arduous, with many person-months of effort invested in several of the other proposals, including relatively detailed analyses of implementation effort, realistic examples of use, and vigorous debates of the pros and cons. The fact that a consensus was eventually reached depended in a large part on the shared fundamental design philosophy, both at the high level, such as simplifying the work for the user, reducing the need for arbitrary decisions, and remaining consistent with other analogous choices to the lower level such as ensuring that inter-unit dependencies are fully captured in the name and context clause of a compilation unit.

Conclusion

Although the mutual dependence problem was probably the most difficult design problem we faced, there were many other problems where a number of alternative approaches were proposed as possible solutions. In each case, we debated the alternatives vigorously, but ultimately a consensus emerged, shaped by the criteria provided by our strong, shared design philosophy. Safety, clarity, consistency, ease of use, and efficiency of implementation provided strong criteria that allowed us to select among the competing proposals with a feeling of satisfaction in the end that we had chosen a clearly superior approach rather than just settling arbitrarily for one of many equivalent alternatives.

Although it is conventional wisdom that a design-by-committee generally produces a set of compromises that leave everyone somewhat unhappy, the Ada 2005 design process left its design committee with an unusual level of satisfaction and sense of accomplishment. It seems clear that this outcome was largely a result of the long history behind the committee. This history enabled us to debate vigorously, but we all feel very good about the final result. We had been able to tie our decisions to criteria that we all shared and which we agreed were the key to the unique safety and productivity of the Ada language.◆

About the Author

S. Tucker Taft is the founder of SofCheck, Inc., which develops tools for automating software quality improvement. From 1990 to 1995, Taft served as the lead designer of the Ada 95 programming language. In 2000/2001, he led the development of the J2EE-based <Mass.gov> portal for the Commonwealth of Massachusetts. Since 2001, Taft has been a member of the International Organization for Standardization Rapporteur Group developing Ada 2005.

11 Cypress DR
Burlington, MA 01803
Phone: (781) 856-3344
Fax: (781) 750-8064
E-mail: tucker.taft@sofcheck.com

Maintaining Sanity in a Multilanguage World

Val C. Kartchner
309 SMXG/MXDDA

The Ada 2005 standard will help many users. But the reality of working in a frozen, legacy development environment needs to be addressed. Development in a mixed version (Ada 83 and Ada 95) and mixed language (C and C++) environment involves dealing with many issues. This article addresses the issues that we encountered when developing applications for the Air Force Mission Planning System. These issues fit into three main categories: dealing with Ada strings, using inter-language interfacing, and using different Ada compilers (83 and 95) but maintaining one code base. This article discusses several of the technical issues involved in interfacing Ada, C, and C++ from both a syntactical and run-time perspective.

The Air Force uses three different mission planning systems (Mission Planning System [MPS], Portable Flight Planning Software [PFPS] and Joint Mission Planning System [JMPS]) to plan routes and missions for training and actual war fighting. The Overlay Import/Export Tool (OIET) and MPS Common Route Definition Import/Export Tool (MCIET) were developed to import to the MPS data from PFPS and JMPS, or export data from the MPS for use on PFPS and JMPS. OIET imports/exports machine and user generated graphics needed for flight planning. MCIET does the same for route information. This article addresses some of the obstacles faced during the development and sustainment of these software programs in mixed development environments.

The MPS was developed by a government contractor in the early 1990s. Because mission planning is such a critical task, stability is very important, so the development and run-time systems were frozen at that time, with the exception of some critical upgrades, like the Y2K patch. However, legacy hardware and software are not supported forever. A government contract with Sun provided for the continued availability of the old versions of the operating system and compilers. But new hardware was not easily compatible with the older operating system, and the older hardware was becoming increasingly difficult to maintain. Therefore, a decision was made in 2004 to upgrade to the current hardware, operating systems, and compilers.

New hardware and software allows a great leap forward for both the developers and users. But the criticality of mission planning prevents the users from moving until their entire planning environment is moved. Each aircraft system (i.e., B-2, B-52H, F-117) has a different mission planning environment (MPE) that consists of the core software (operating system and applications) along with additional installable software modules (ISMs) and other software. This means that the OIET and MCIET need to be maintained for both the old system and the new system until all aircraft MPEs have been upgraded to the new system.

String Issues

Because many of Ada's features were originally designed to address Department of Defense (DoD) needs, Ada handles several important features (such as strings, pointers, and memory allocation/deallocation) differently from its programming peers – languages such as C, C++, and Java. This article covers several important topics that merit separate discussion, especially in the context of not only multiple-language programming, but also in the context of a mixed-mode language environment. One of the most important issues involves the way Ada treats strings as opposed to other languages.

Ada Strings

The initial MPS development environment had been frozen with an Ada 83 compiler. Strings in Ada 83 (as in most computer languages) are arrays of characters. Ada 83 was designed with a great consistency about how arrays are handled: arrays must be defined with a particular size before they can be used; when assigning one array to another, they must be the same length and contain the same type of data. This works fine for an array of generic data but not nearly as well for strings where variable-length is normal.

When strings are passed around, variable lengths are often used. Receiving a variable-length array (including strings) as a parameter to a subprogram (procedure or function) is relatively easy to do. However, returning a variable-length array (or string) as the return value of a function or an *out* parameter of a procedure is not possible using standard Ada 83 language constructs. Some sort of substitute (or trickery) must be used to accomplish the desired result. This is why the Ada 95 standard added the unbounded-string type and the allocate-and-assign construct (*pointer: = new string'(trim_ spaces (some_ string));*), so that standard solutions will be available.

But since we needed to pass variable-length strings in Ada 83, we developed

Figure 1: *Core of VString Package*

```
package VString is
  type VString is limited private;
  subtype C_String_Ptr is CString_Interfaces.C_String_Ptr;
  …
private
  type String_Access is access String;
  type VString is
    record
      cur_length : Natural := 0;
      str_access : String_Access := Null;
    end record;
end;
```

the VString (variable string) package. A VString variable is basically a string-access (string pointer) with the current-length-used, but the VString package hides the implementation details (see Figure 1). This package also provides an interface (supporting procedures and functions) to do what is needed to VString variables. The VString type could have been implemented as a private type, but because of the string access type inside, it is not recommended.

For instance, if VString is a private type and two variables (*Aye* and *Bee*) contain the VString values of *Alphabet* and *Spelling Bee* respectively assigning Aye to Bee (or *Bee := Aye;*) would copy the values exactly (i.e, *Bee.cur_length := Aye.cur _length;* and *Bee.str_access := Aye.str_access;*) The access to the *Spelling Bee* string is now lost. This is called a *memory leak*[1]. The string reference by both *Aye* and *Bee* is also referenced (referenced two or more times) so that if *Aye* is set to *Numerics* (using VString functions), *Bee* also holds the same string. This is not usually desirable.

Also, if *unchecked_deallocation* is instantiated to free (return to available memory) the string-access memory, and both *Aye* and *Bee* are freed, the same memory will be freed twice, creating a potentially disastrous problem in the program.

This is why VString is declared as a *limited private* type, so that assignment is not allowed between two variables of the same type. With Ada95, these problems do not exist because limited-private types may have the assignment operator overloaded and tagged-types have the equivalence of destructors (through *Ada.Finalize*).

Support subprograms are used to copy Ada strings or C strings to VStrings. VStrings may also be copied to Ada strings, with automatic truncation or extension (space filling) to the size of the destination string. A VString is also kept null-terminated[2] so that it can be passed to a C or C++ function. As needed, other support subprograms to compare, paste, slice, search, replace, output, and debug, have been added to the package. This package has proven useful and is used extensively in OIET and MCIET.

There are some good reasons for allocating more character space (in the string) than will be immediately needed. Dynamic memory allocation algorithms will always allocate memory in multiples of a *minimum allocation unit*, so taking advantage of this does not consume more actual memory than usual. Also, depending on the program and string, a string may change size several times over its lifetime and allocating a little more memory is likely to delay the need for reallocation of memory as the string grows. If the length of the string decreases, it may only be a temporary decrease, so holding on to the memory is not likely to adversely affect performance. When Ada calls for the actual string, the slice of the allocated string that is in use is returned.

Because a limited private type cannot be automatically copied (in Ada 83), we had to find another way of including variable-length strings in nodes of a generic linked list package. Instead, we used string access types directly in the nodes of the list. Another option would have been to provide a copy procedure as one of the parameters to the generic linked list package.

This VString package worked for us because we had to use Ada 83. But if you have the option, use the improved string handling capabilities added in Ada 95 or Ada 2005.

> *"Another key to interfacing with another language is understanding how each language will pass subprogram parameters."*

More on C Strings

Ada83 has no standard way to pass or receive C strings, but the compiler that we used provided a proprietary method of doing this. The Ada95 standard has defined the *Interfaces.C.Strings* package as this interface. Passing and receiving C strings is done extensively throughout the Ada portion of our code; using the respective compiler-provided interfaces would split our code into pairs of files that would be mostly identical, thereby increasing maintenance costs. Also, using a file preprocessor would have been unwieldy in this case.

The least disruptive method to pass and receive strings was to create two different packages, one for each Ada (83 and 95) compiler. Each package would reside in a different file but internally call itself *CString_Interfaces*. Each package would present the same interface to users of the package and become an intermediary to the actual functionality provided by each compiler. Relatively small changes were then made to the code that needed to interface with C strings (including the VString package discussed above). One of these changes was to use *C_String_Ptr* wherever a pointer to (address of) a C String was needed. Any user of this package can use *C_String_Ptr* without knowing (or caring) what the actual type is. As needed, other subprograms to perform needed functions, like copying to and from C Strings, are provided in the *CString_Interfaces* package. The specification and body for each of the two implementations resides in the same directory and a *make*[3] file is used to compile the appropriate version of the package.

Interlanguage Interfacing

Since some of OIET and MCIET are written in C, strings are also often passed between Ada and C, but Ada strings are not simple. It is easier to pass C Strings using the C-String interface package provided with your compiler, as discussed in the previous section. When passing a string to a C function, unless special considerations have been made (like passing in the length also), each string will need to be null-terminated. To facilitate this, the VString package places an *ASCII.null* after the used portion of each VString. When the *cptr* function is called, it returns a *C_String_Ptr* ready to be passed into a C function. Like the *C_String_Ptr* used, it is good practice to declare a new type for each pointer type that is passed around so that a user does not accidentally pass a pointer of one data type to a function expecting another.

Also, do not assume that an *integer* in Ada will correspond to an *int* in C. Depending on the compiler and compiler options, it may or may not. But finding out which types of addresses, integers, and floating-point values correspond between Ada and C is a simple process of consulting the respective compiler manuals.

Another key to interfacing with another language is understanding how each language will pass subprogram parameters. FORTRAN always passes parameters *by reference*. That is, if *1* is passed to a subprogram, the value of *1* is placed in memory and the address of the memory location is passed to the receiving subprogram (in a processor register or on the processor stack). C passes parameters *by value* or *by address*. Passing by value means that if a value of *1* is passed as a

parameter; a *1* is passed to the receiving subprogram. Passing by address means that the same thing happens as passing *by reference*. C++ may pass *by value*, *by reference,* or *by address*.

The difference between *by address* and *by reference* is determined by how the receiving program treats the address received. If the parameter is received *by reference*, then the receiving subprogram knows to fetch the value from the address specified. That is, the address of the variable is implicitly *dereferenced*. If the subprogram receives *by address*, it must specify that it is fetching the value from the address. That is, the address of the variable is explicitly *dereferenced*. This applies to Ada when an access type is passed into a subprogram and .all is used to get the value stored at that access location.

It is important to understand the different ways that parameters may be passed to discuss how to interface between languages. In both of the Ada compilers used for this project (Ada 83 and Ada 95), a parameter specified as *in* is passed *by value*, and a parameter specified as *out* or *in out* is passed and received *by reference*. Simple types (integer, floating-point numbers, addresses, etc.) are returned from Ada and C functions *by value*. Ada may return complex types (strings, records, arrays, etc.), but we did not experiment with returning such types between languages.

Concerning subprogram parameters, Ada is more restrictive than C. Ada procedures allow parameters to be *in*, *out*, or *in out*, but allow no value to be returned. Ada functions have return values, but parameters can only be *in*, which is also the default if not specified. It is best to use the more restrictive Ada rules. In C or C++, the equivalent to an Ada procedure would be a *void* function. If a C function needs to have both *out* parameters and a return value (such as a system function), then a wrapper function can be used.

For instance, to find the status of a file, the system function *int stat(const char *path, struct stat *buf);* is used. Success or failure status is returned from the function as an integer value, and the status of the file itself is returned in the buffer. A C wrapper function could look like this: *void wrap_stat(int* result, const char *path, struct stat *buf) { *result = stat(path, buf); }*. The Ada declaration to call this C function would look like *procedure wrap_stat(result : out integer; path: in C_String_Ptr; buf : out stat_record_type);* with the supporting declarations of the *stat_record* and C function.

Ada and C++ Issues

One problem between Ada and C++ surfaces during runtime. Both Ada and C++ have variables, records, and objects that must be initialized when the program starts to run, before the first subprogram (*main* in the case of C++) is entered. In order to do so, each language wants to control program start-up, but only one can. An alternative considered was to let one of the languages start the program then call the other language's *initializer* (subprogram to initialize data). This varies from compiler to compiler and even operating system to operating system. These routines were not found in the manuals for our compilers, so another plan had to be devised.

The program was broken into two pieces: one program initialized by Ada and one program initialized by C++. The Ada portion runs the Graphical User Interface (GUI) and calls the C++ portion much like a subroutine would be called. This is implemented by using the Unix system functions *fork* (to start another child process), *execl* (to execute a new program inside of a process), and *wait* (to wait for a child process to complete). The exit status of the subprogram is used where a return value would normally be used, but the type may only be an unsigned integer in the range of zero to 255. To use this value returned from the child process, symbolic names (constants) are defined for each of the return values used in both Ada and C++ to indicate what type of success or failure has occurred. For instance, the value of zero is defined as *STATUS_SUCCESS* or fully successful completion. For longer messages meant for the user to view (error, warning, or informational messages), the name of a log file is passed as a parameter to the new program. If there is anything in the log file to show the user, an appropriate value is returned. In the case of abnormal termination of the child process, a system error code is automatically returned by *wait*.

Some code needed for results in the GUI had already been written in C++, and by its nature could not easily be segmented to run as another program. Through experience it was found that the C++ Standard Template Library (STL) relied on the *initializers* being called, so they could not reliably be used in any of these routines. However, while resolving this issue, it was also discovered that even though the C++ *initializers* are not run, global and static memory for simple types (ints, chars, pointers, arrays, structs, etc.) was initialized as expected, but the memory occupied by global and static objects (instantiations of classes) is always cleared, initialized to zeroes. There was a need to program defensively, but the clearing of the memory could be used advantageously. If a

Figure 2: *Exception Handling Example for Ada 83*

```
--Top of file (For Ada 83 compiler)
With Current_Exception;
…
-- Exception block
exception
  when others =>
    error_message("Exception " &
        Current_Exception.Exception_Name &
        " propagated out of Export_Overlay");
end Export_Overlay;
```

Figure 3: *Exception Handling Example for Ada 95*

```
--Top of file (For Ada 95 compiler)
With Ada.Exceptions;
…
-- Exception block
exception
  when Event: others =>
    error_message("Exception " &
        Ada.Exceptions.Exception_Name(Event) &
        " propagated out of Export_Overlay");
end Export_Overlay;
```

key variable was still zero, then the object had not been initialized at compile time so it had to be initialized at run time. This may not be true of other compilers, but it is of both of the Ada compilers used for OIET.

There are additional problems when interfacing between Ada and C++. Both languages allow subprogram overloading[4] (including operators), but must work with linkers that require unique symbol names when assembling the program from the separately compiled source files. In order to do this, each compiler makes unique symbol names by *name mangling* (changing the name) but in different ways. For instance, two Ada functions *function calc(it: integer) return integer;* and *function calc(it:float) return float;* in the package *test* become the linker symbols *_A_calc.3S10.test* and *_A_calc.4S10.test* respectively. The equivalent functions declared in C++ become the linker symbols *_Z4calci* and *_Z4calcf* respectively. These exact linker symbols will be different depending on the compilers used.

C does not allow overloading so the function names are not changed to accommodate this feature. Both Ada and C++ provide for linking with code written in the C language. The simplest way to resolve the name mangling issue is to indicate to each compiler that it will be interfacing with C even when there will be no intermediate C function. For C++, the function will also have to be a global function or a static class method.

Different Ada Compilers, One Code Base

Between the Ada 83 and 95 compilers, there are differences in how some language constructs are specified. In the Ada 83 standard, there were suggestions on how to use *pragma* statements to export Ada symbols for use by other languages and how to import symbols from other languages for use by Ada. The Ada 95 standard specified how these interface *pragma*s are to be. But the standard way is different than how our Ada 83 compiler implemented them.

Also, when catching an *OTHERS* exception, Ada 83 provides no standard way to find out what specific exception has been caught. The compiler that we used has a set of functions that will retrieve the exception name and the procedure in which the exception occurred so that we can notify the user (through the GUI) of the problem (see Figure 2). Ada 95 provides a standard way to do this, one different from the proprietary method our Ada 83 compiler used to implement this feature (see Figure 3).

```
#ifdef C2_2d
With Current_Exception;
#define EXCEPTION_EVENT(x)  x
#define EXCEPTION_NAME  Current_Exception.Exception_Name
#else
#ifdef LCU
With Ada.Exceptions;
#define EXCEPTION_EVENT(x)  Event: x
#define EXCEPTION_NAME  Ada.Exceptions.Exception_Name(Event)
#else
#error "Must define Core version number"
#endif
#endif
```

Figure 4: *Significant Portion of "exceptions.a"*

```
--Top of file (Before preprocessing)
#include exceptions.a
…
-- Exception block
exception
  when EXCEPTION_EVENT(others) =>
    error_message("Exception " &
        EXCEPTION_NAME &
        " propagated out of Export_Overlay");
end Export_Overlay;
```

Figure 5: *Exception Handling Example for Preprocessor*

Both the exception differences and the interface *pragma* differences are issues with parts of the language so they could not be fixed by using an intermediate package. In C or C++, these types of differences would usually be handled by using preprocessor directives to perform conditional compilation and/or macros. The Ada 83 compiler that we are using has a proprietary preprocessor similar to the C preprocessor but using Ada-like syntax. The Ada 95 compiler provides nothing like a preprocessor. Several other solutions were considered, but it was decided that using a preprocessor would be the simplest to implement. To illustrate what was done, the exception example will be used because it is slightly more complex.

The file preprocessor included with a C compiler is simple in theory, substituting text and macros where the preprocessing symbol appears. But when the GNU C compiler was used with the *preprocess only* directive, it objected to the Ada code surrounding the preprocessor code. The result was the same for the GNU C++ compiler.

Brief consideration was given to writing a preprocessor to meet our needs, but to do it right seemed like a two to three week task. In the hope that someone else had encountered a similar problem and already crafted a solution, a search was undertaken at the two largest open-source repositories on the internet: <www.sourceforge.net> and <www.freshmeat.net>. After exploring a few of the resultant programs, *filepp* was found on the later site. It turned out that it does exactly what is needed but is also highly configurable (in case it does not do what is needed)[5,6].

To use the preprocessor to do what needs to be done, a file *exceptions.a* was created that contains the substitutions that need to be done. This file is written such that if no version is specified, the preprocessor will display an error (see Figure 4). To use this file, an *#include exceptions.a* is placed at the beginning of the file where the other *with* directives occur. The exception-handling portion of the code is then rewritten to use the macros (see Figure 5). For instance, *EXCEPTION_EVENT(x)* is a macro expecting text between the parenthesis when used. This text will then be placed where x appears in the text at the end of the line (compare Figures 2 through 5).

If this were used on the file *test.a* to produce the preprocessed version of the file (*test_p.a*) for the Ada 95 compiler (that we call Life Cycle Upgrade [LCU]), the command *filepp -DLCU -o test_p.a test.a* would run. Again, this would be placed in the make file so that the preprocessing is automatically performed

on *test.a* when the program is built. The Ada compiler receives the file *test_p.a*.

Conclusion

The reality of legacy development environments and systems is that not all programming problems can or should be accomplished in the same language, development environment or even the latest versions of these tools. The real world just is not that simple. And sometimes we have to choose the best tool for the job from those available. Interfacing with one or more other languages also requires knowledge of data representations and how the languages send and receive the data. Solutions can be found that will allow maintaining the same code base on two (or more) disparate operating environments without too much maintenance overhead.◆

Notes

1. Both the Ada 83 and Ada 95 standards allow for automatic garbage collection but do not require that the memory be reclaimed until after the type goes out of scope (see <www.adaic.org/docs/craft/html/ch11.htm> and <www.adaic.org/docs/craft/html/ch11.htm>). Since VString is declared in a package, it will not go out of scope until the program exits.
2. C and C++ use the ASCII null character as a sentinel to mark the end of strings. C++ has also defined a more Ada-like string type as part of its standard.
3. *Make* is a program commonly used to build programs from the component source files. The *make file* or *makefile* describes to make the order and how to process each file to make the final result.
4. *Overloading* is having two (or more) subprograms with exactly the same name but different parameter types. The compiler determines which subprogram to call by the types of the parameters passed.
5. See the documentation at <www.cabaret.demon.co.uk/filepp/>.
6. The substitutions done by *filepp* are case-sensitive. In the case of these files, the C convention of making preprocessor symbols all uppercase is used.

About the Author

Val C. Kartchner is the lead programmer of the Overlay Import/Export Tool installable software module for the Air Force Mission Planning System. He has more than 15 years experience in software development, design, and maintenance at Hill AFB, the last six years of which is directly for the Department of Defense. This experience in several different programming languages and development environment has granted the experience necessary to solve issues such as those detailed in this article. Kartchner has a Bachelor of Science in computer science from Weber State University.

309th SMXG/MXDDA
6137 Wardleigh RD
Hill AFB, UT 84056-5843
Phone: (801) 775-2777
Fax: (801) 775-2772
E-mail: val.kartchner@hill.af.mil

Web Sites

ACM SIGAda

www.acm.org/sigada

Here you will find information on the special interest group (SIG) Ada organization and pointers to current information and resources for the Ada programming language. It is a resource for the software community's ongoing understanding of the scientific, technical, and organizational aspects of the Ada language's use, standardization, environments and implementations. This is ACM SIGAda's latest effort to help expand accessibility to Ada information. They want to make this the one stop for information on both SIGAda's current activities and on the Ada language and community at large.

The Ada Information Clearinghouse

http://adaic.org/

The Web site provides articles on Ada applications, databases of available compilers, current job offerings, and more. The Ada Information Clearinghouse is managed by the Ada Resource Association, a group of software tool vendors who support the use of Ada for excellence in software engineering.

Ada Home

www.adahome.com

Since March 1994, this server provides a home to users and potential users of Ada, a modern programming language designed to support sound software engineering principles and practices. The Ada Home Floors and Rooms contain many unique tools and resources to help expand knowledge and increase productivity.

Ada World

www.adaworld.com

Ada World has been created essentially to bring the Ada programming language a central place where Ada developers and curious programmers can learn about Ada, see what is happening as far as Ada development projects go, and give a good idea of what can be done with Ada. To reach this goal, Ada World serves as a place where Ada developers can talk about Ada as well as work on development projects.

Ada Power

www.adapower.com

Ada possesses the ultimate in flexibility (oo and non-oo), real standardization, and validation, true cross-platform programming, incredible compile time error checking, readable code, and support of all levels of software engineering. As a way of contributing back to the Ada community and to help advocate this powerful language, AdaPower.com was formed, and includes on its Web site examples of Ada source code that illustrate various features of the language and programming techniques, various interfaces to popular operating systems (thick and thin level bindings), and examples of Ada source code that illustrate various algorithms; a collection of packages for reuse in Ada programs; and articles on implementing software in Ada.

Adapting Legacy Systems for DO-178B Certification

Paul R. Hicks
AVISTA Incorporated

The avionics world is moving toward greater integration of avionics products used in both commercial and military aircraft. Document Order (DO)-178B certification is now being required in some areas in the military (such as military aircraft flying in European civil airspace), and may be considered in others. It is possible to achieve a cost effective approach to enable legacy systems to meet DO-178B certification requirements by performing a gap analysis to determine what existing activities and artifacts can be reused for DO-178B certification and define the remaining tasks that need to be completed in order to fulfill certification requirements.

RTCA Document Order (DO)-178B [1, 2] is a long-used standard mandated by the Federal Aviation Administration (FAA) for the certification of commercial airborne avionics systems containing embedded software. In recent years, DO-178B is also being applied, at least in principle, to some non-commercial avionics systems. Many of these systems may have been developed with other standards in mind. However, aspects of DO-178B are being applied where certification will be enforced in the future. The level of compliance with DO-178B is typically influenced by budget and schedule constraints. In an increasing number of instances, the military sector is at least considering DO-178B certification.

While DO-178B may be viewed as an eventual requirement for all airborne avionics systems, commercial and military systems alike, it is currently evaluated on a case-by-case basis for military programs. The impact of incorporating DO-178B requirements on a program does not come without significant impact to budget and schedule. This impact varies depending upon the software level (A through E) imposed on the application and is a critical factor in the decision to pursue certification (see Figure 1).

Other critical factors in determining the impact of cost (budget and schedule) include the size and complexity, the system, and the maturity of the procedures and processes utilized by the software development and verification teams. Companies with more mature processes institutionalized across their organization will be able to adapt much more effectively and efficiently.

Impact to Budget and Schedule

While not yet a requirement for every military avionics system, there are some programs that do impose DO-178B certification. In this scenario, the cost and schedule impact certainly needs to be accounted for and minimized. When DO-178B certification is not imposed as a requirement, the impact to the cost and schedule must be measured against the benefits gained. The argument that DO-178B adds significant quality to a legacy system may be disputed when examining the service history of an avionics system that has countless hours of flight time. However, DO-178B processes may help identify potential deficiencies in requirements definition and/or testing by performing structural coverage analysis. In this scenario, it may be difficult to justify the budget and schedule impact when DO-178B is not an imposed requirement.

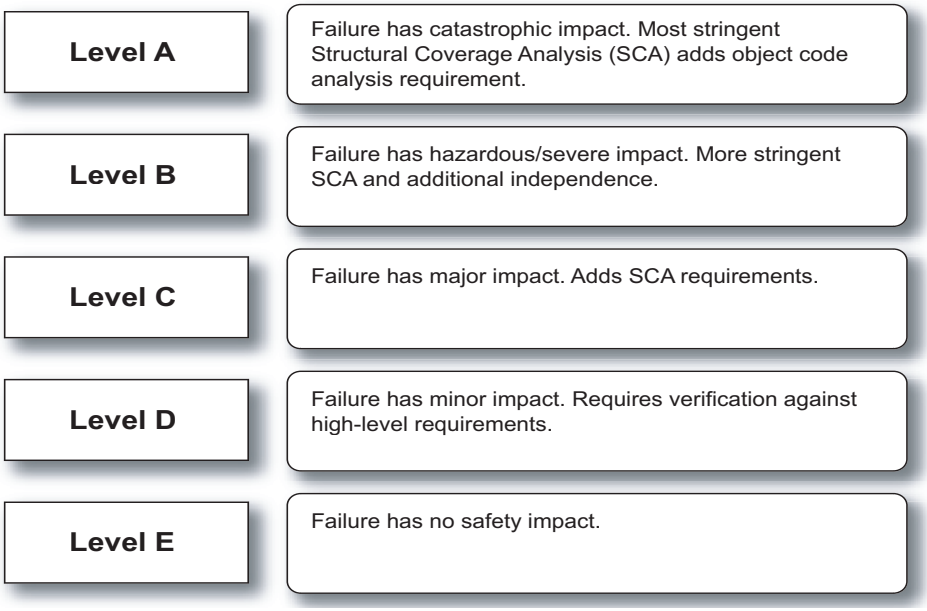
While the requirement to satisfy the criteria outlined in DO-178B may appear to be a daunting task for the engineering teams who maintain legacy military avionics systems, the effort of adapting the legacy system may be easier (and cheaper) than originally perceived. The key is to accurately estimate the impact to budget and schedule. While it is easy for engineering teams to underestimate the budget and schedule impact, it is also possible to overestimate the impact by not taking advantage of existing processes. To accurately estimate the impact, companies can and should take advantage of their existing planning documents and testing processes.

Value in Legacy Systems

There are significant benefits for a legacy avionics system to incorporate the objectives outlined in DO-178B. Best practice concepts have been derived by key members of the aviation community through implementing the certification process. These best practices continue to be refined and enhanced based on increased use, evolved technology, and gained experience as evidenced by the evolution of DO-178B.

The DO-178B specification enforces good software engineering practices by providing guidelines for the production of

Figure 1: *DO-178B Certification Levels A Through E*



embedded software for airborne systems. These guidelines ensure that the systems perform their intended function with a level of confidence in the safety of the system. DO-178B serves simply as a guideline outlining the objectives to be met, the activities to be performed, and the evidence to be supplied.

DO-178B does allow for alternate methods for satisfying one or more objectives [3]. These alternate methods can be used in lieu of some of the more typical methods described throughout DO-178B requirements. However, alternate methods are more of an art than a science. There are several dependencies associated with any of these alternate methods, and there may or may not be opportunities to pursue these alternate methods. If you are considering an alternate method, consult with a Designated Engineering Representative (DER) [4, 5] with experience in the particular alternative method. With that said, the focus of this article describes using a more traditional approach.

Gap Analysis

One common misperception is that very few artifacts can be reused to upgrade a non-DO-178B certified legacy system to a certified legacy system. A start-from-scratch approach is too often the first thought to retrofit DO-178B guidelines within a legacy system. Misunderstanding the scope of a project often leads to wildly inaccurate estimates with regards to the costs and schedules associated with elevating an application to the DO-178B standard. Individuals who are best qualified to perform a gap analysis should know the specific requirements for each software level of DO-178B certification, understand the existing processes of the legacy system, and have the authority to make decisions.

To accurately estimate the associated costs and schedules, we recommend that you follow these steps while performing a gap analysis (see Figure 2).

Step 1: Determine the software level (level A through E) that should be assigned to your application. The software level is determined by the severity of the failure conditions on the aircraft and its occupants. This is typically identified by performing a system safety assessment as described in DO-178B [1]. The software level may be predictable based on the functionality of the application with respect to similar industry applications.

Step 2: Understand the guidelines identified in DO-178B. Those who have not been involved with these certified systems before can find the learning process overwhelming. However, project teams that are new to DO-178B can learn from the several companies and organizations in the industry that have acquired a breadth of related experience.

The tables in Annex A of DO-178B [1] summarize the software life cycle process objectives and outputs by software level. These tables can serve as the foundation for your gap analysis to determine which activities are required to comply with DO-178B.

Step 3: Determine what activities have already been accomplished and how they can be applied to the guidelines identified in DO-178B. Many companies have solid software development processes and procedures already in place. Even though the software engineering activities that were performed may not have focused on DO-178B, these practices provide the most likely opportunities for reuse if the foundation behind the processes followed were built on solid software engineering practices – whether driven by other industry standards, industry certifications (such as the Software Engineering Institute's Capability Maturity Model® Integration [CMMI®] or International Organization for Standardization [ISO] 9001), or good engineering judgment. Credit for much of the effort previously performed can be used for activities and artifacts identified in DO-178B. Refer to the sidebar for an example of specific activities and artifacts that can be applied to guidelines identified in DO-178B.

Step 4: Take advantage of existing processes currently employed that fully or partially achieve compliance to DO-178B. It is generally not cost-effective to reinvent the wheel. The project team should supplement existing processes wherever possible. However, it is not cost-effective to utilize every existing process, especially if the process is not a useful activity to attain DO-178B certification. Consider eliminating processes not directly related to certification, or replacing these ineffective processes with more efficient ones.

By following these steps, your project team should be able to establish which objectives are completely satisfied, which objectives are partially satisfied, and which objectives are completely unsatisfied. The list of activities and artifacts identified within your gap analysis may vary with each company. If solid practices and processes are consistently implemented, fewer deficiencies will be identified in your gap analysis. If the practices and processes are uniformly institutionalized across the company, the deficiencies identified in the gap analysis should be similar across different product lines with the same software level.

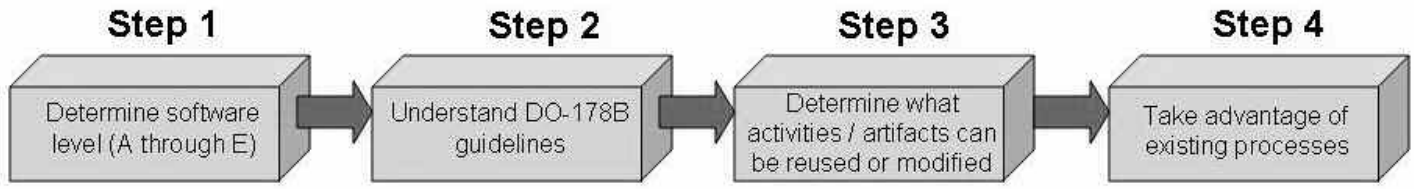
Common Deficiencies

The lack of certification and planning documents are typical examples of deficiencies; specifically, documents necessary for certification submittal. These documents are the Plan for Software Aspects of Certification to describe your certification plan, and the Software Accomplishment Summary to illustrate compliance with your certification plan. If planning documents do exist, they often must be modified to ensure that they address the content described in DO-178B.

If you are using an implementation-based testing approach, the conversion to requirements-based testing could be costly and time consuming. DO-178B endorses a requirements-based functional testing approach, where your test cases and procedures are based upon the software requirements data. If you use this testing approach, you will be able reuse your original verification test suite. You probably will need to enhance your requirements-based test suite to ensure that the requirements are completely tested. In addition,

® Capability Maturity Model and CMMI are registered in the U.S. Patent and Trademark Office by Carnegie Mellon University.

Figure 2: *Gap Analysis in Four Steps*



requirements coverage analysis must be performed to ensure all requirements have been sufficiently addressed.

Another example of a deficiency is in the area of structural coverage analysis. The majority of the systems developed outside DO-178B do not perform structural coverage analysis. Performing structural coverage analysis ensures that all software constructs have been exercised by the requirements-based test suite. Software constructs that have not been exercised are used to identify inadequacies in the software requirements, shortcomings in the requirements-based test cases and procedures, deactivated code, and/or dead code. Each shortcoming must be resolved or justified.

Traceability

An area that is occasionally overlooked is traceability. Traceability is used to illustrate evidence of an association from an output to its origination. Typical traceability activities may include the following types:

- Requirements traceability from the lower-level requirements to the higher-level requirements.
- Source code traceability from the source code to the lower-level requirements.
- Test case and procedure traceability from the test cases/procedures to the lower-level requirements.

The goal of traceability is to be able to follow a continuous thread throughout the entire product life cycle to confirm the link between the requirements data and its associated source code and tests cases and procedures.

Independence

Certain objectives of DO-178B also require independence. Independence is the separation of responsibility between the developer and verifier to ensure no implied biases are applied to the objective under review. The objectives that require independence vary with the software level imposed on the system. It may be worth considering applying independence wherever feasible.

Configuration Controls

Some outputs of DO-178B have established configuration management controls imposed on them. Control categories define the configuration management control placed on each data item. The control category placed upon a data item also varies with the software level imposed upon the system. Again, it may be worth considering applying the more stringent configuration controls wherever feasible.

Adapting a Non-DO-178B Certified System to a DO-178B Certified System

Consider the example of a legacy Global Positioning System (GPS) portion of an inertial navigation unit, in which the system would be required to upgrade to a DO-178B certified system in the future. The engineering team responsible for the upgrade assumed that they would have to start over. But, by performing a gap analysis, they were able to decrease the cost by a factor of six by taking advantage of existing activities previously performed and existing legacy artifacts, such as planning documents, requirements data, code, test cases and test results. They were able to reuse their requirements-based testing procedures, and the system already had good processes in place because they had adopted Software Engineering Institute Capability Maturity Model Integration Level 5 processes when developing the original GPS software.

Once you have established the deficiencies found in the gap analysis, the next step is to formulate a plan to resolve the deficiencies.

Efficient Planning

As discussed earlier, the effort to obtain the DO-178B certification does not come without cost, effort, or risk. Even organizations with previous DO-178B experience still experience unexpected pitfalls – not unlike any software engineering effort. A commitment from the stakeholders is required in order to be successful.

With the information collected during the gap analysis, you can then establish a task list. Based on the findings in the gap analysis, some of the tasks may be obvious and estimates can be easily applied. For example, gathering the structural coverage from the existing test suite can be fairly straightforward. However, there may be some tasks that cannot be easily estimated until additional fact finding efforts are completed. For example, to achieve complete coverage, the effort to supplement the requirements data and test suite are strictly dependent upon the results of the structural coverage analysis effort. For this reason, you may want to consider a phased approach.

Phased Approach

While a DO-178B requirement may not yet have been imposed, start planning early if there is an expectation that it will be imposed in the future. If you employ a phased approach, there are several benefits that can be realized such as the following:

- The costs may be spread out over multiple fiscal years, easing the financial impact.
- By spreading the work over a large time span, you can utilize a smaller engineering team.
- The higher risk items can be performed earlier so that the risk can be mitigated or addressed in advance of the deadline.
- Activities that lead to better estimates for follow-on activities can be performed earlier so that the follow-on activities can be more accurately estimated in advance of the deadline.
- The team has the opportunity to learn process changes earlier, thus gaining familiarity and more insight to realize process improvement opportunities.
- There is a longer history of subjective evidence to support the project.

Using a phased approach enables you to react more quickly and more effectively when the DO-178B requirement is imposed. It reduces the risk of having to quickly assemble a large team for the project at the last minute.

DERs

Involve a DER or equivalent early in the determination process. A DER is an independent specialist and an experienced engineer designated by the FAA as having authority to sign off on your project as a representative of the FAA [5]. You should establish a solid plan and have the DER approve your plan as early as possible to confirm your approach. In addition, make sure that you execute to the plan. You are not restricted from deviating from the plan when and where it makes sense. However, the deviations must be communicated to the DER as they are identified to ensure approvals. The more familiar the DER is with the plan and your execution of the plan, the more likely it is to receive the final acceptance of the certification package. If you do not have a DER on staff within your company, there are independent DER consultants that your company can hire to work with you.

Conclusion

It is important to understand that cost (both budget and schedule impact) is a significant factor that often prevents organizations that supply avionics systems from providing fully DO-178B compliant software when not required. While there are

ways to reduce the cost impact, history has shown that the cost generally prohibits the implementation of DO-178B compliance when it is not a requirement. However, the industry is trending towards some level of DO-178B consideration. When it becomes a requirement, cost can be minimized by taking credit for activities and artifacts already incurred and by establishing cost effective and efficient approaches to achieving DO-178B compliance.

Converting non-DO-178B legacy systems to comply with DO-178B guidelines will become more common as requirements such as the Global Air Traffic Management program begin to enforce DO-178B certification on all avionics systems that share the world's airspace. Do not wait until that day happens; get a head start by integrating DO-178B within your legacy systems now.

By performing a rigorous gap analysis, your project team will be able to accurately access the cost and schedule involved in developing and implementing a plan for your legacy system to receive certification. Bring in a DER early on in the development process to ensure final acceptance of DO-178B certification.◆

References

1. Radio Technical Commission for Aeronautics. "Software Considerations in Airborne Systems and Equipment Certification." 1 Dec. 1992. <www.rtca.org>.
2. Radio Technical Commission for Aeronautics. "Final Report for Clarification of DO-178B for Software Considerations in Airborne Systems and Equipment Certification." RTCA/DO-248B. 12 Oct. 2001 <www.rcta.org>.
3. Certification Authorities Software Team Position Paper, CAST-5. "Guidelines for Proposing Alternate Means of Compliance to DO-178B." June 2000 <www.faa.gov/other_visit/aviation_industry/designers_delegations/designee_types/der/>.
4. "Avionics Software." Science Daily <www.sciencedaily.com/encyclopedia/avionics_software>.
5. Federal Aviation Administration. "FAA Consultant DER Directory." <www.faa.gov/other_visit/aviation_industry/designees_delegations/designee_types/media/DERDirectory.pdf>.

About the Author

Paul Hicks has more than 18 years of experience leading avionics software engineering programs for DO-178B certification. As a senior programs manager for AVISTA Incorporated, Hicks has worked on a variety of avionics systems including primary flight displays, flight management systems, autopilots, and planning and execution of adapting efforts to upgrade legacy products for DO-178B certification. His specialty is in avionics systems engineering. Hicks has a bachelor's degree in computer science from the University of Wisconsin, Platteville.

AVISTA Incorporated
PO Box 636
1575 E Business HWY 151
Platteville, WI 53818-0636
Phone: (608) 348-8815
Fax: (608) 348-8819
E-mail: paul.hicks@avistainc.com

Is There Anything Better Than Utah in August?

Try attending 4 world-class seminars for the price of one!

Seminars include:

- 7 Habits of Highly Effective People Signature Course – Franklin Covey
- Crucial Conversations – VitalSmarts
- The Choice at Work – Arbinger Institute
- Coaching Essentials – Based on Jim Collins' best-selling book "Good to Great"

Date: August 22-31, 2006
Time: 8:00 a.m. - 5:00 p.m.
Location: Hill AFB, Utah
Cost: $2,875

S T S C

These seminars are the core curriculum of the Hill AFB Leadership Training program.

Contact Debra Ascuena at (801) 775-5778
DSN 775-5778, or e-mail: debra.ascuena@hill.af.mil

Ada: The Maginot Line of Languages

-or-

One language to rule them all, One language to find them,
One language to bring them all and in the darkness bind them.

(with apologies to J.R.R. Tolkien)

During World War I, more than one million French citizens were killed, and another estimated four to five million were wounded. Many French politicians and generals thought that the Treaty of Versailles (which ended the war, and was supposed to punish the defeated countries and prevent further conflict) was insufficient protection. France was justifiably concerned that the treaty was really just an armistice and that war would ultimately resume (as it did – World War II). To protect France, many influential politicians and generals were in favor of an aggressive set of fortifications. There were many studies and meetings, and based on the consensus of opinion, the Maginot Line was built.

The Maginot Line, named after French minister of defense André Maginot, was a line of concrete fortifications, tank obstacles, machine gun posts, and other defenses which were built along the Italian and German border. The French thought that these fortifications would slow down attacking forces, allowing the French time to respond. Two places the Maginot line did not extend were the Ardennes Forest (which was thought impassable) and the Belgium border, because Belgium and France had recently signed an alliance.

When World War II began, the Germans did not view the Ardennes Forest as impenetrable. More than a million troops and 1,500 tanks crossed Luxembourg, Belgium and then moved straight through the Ardennes. On May 10, 1940, the German advance started. The French government had to abandon Paris on May 13. The conquest was swift and decisive.

History has sometimes viewed the Maginot Line as something that was ineffectual. However, this viewpoint, in my opinion, is vastly incorrect. The Maginot Line did exactly what it was supposed to do – prevent a direct attack upon France's Eastern border. The few places upon the Maginot Line that were directly attacked by German troops held out well. The concept was sound, the execution was just incomplete. There is a history lesson to be learned here.

The theme of this issue is Ada 2005. Now, for those of you who don't know me, I'm an Ada zealot. I taught one of the first U.S. Air Force-approved Ada training courses back at Keesler, AFB in 1984. I taught Ada at the Air Force Academy starting in 1986.

Back in the 1980s, there were literally hundreds (possibly thousands) of programming languages running around. Every defense program and contractor used their own language (or variation of a language). Most projects were in assembly language of some type, making projects hard to maintain and upgrade. The initial vision of Ada was to provide a common high-order programming language that would allow Department of Defense (DoD) software that was cheaper and quicker to develop and easier to maintain. Ada can be described as a language that has facilities for real-time response, concurrency, hardware access, and reliable run-time error handling. In support of large-scale software engineering, it emphasizes strong typing, data abstraction and encapsulation. Nothing bad in this list – in fact, everything in this description sounds pretty good, doesn't it? So good, in fact, that back in 1983, Richard DeLauer, then Under Secretary of Defense for Research and Engineering, sent out a memo directing that:

> The Ada programming language shall become the single common programming language for Defense mission-critical applications. Effective 1 January 1984 for programs entering Advanced Development and 1 July 1984 for programs entering Full-Scale Engineering Development, Ada shall be the programming language.

The problem was that back in 1983 there weren't many compilers, tools, or experienced programmers. Compilers were slow and tended to consume all the resources of even high-end computers. The general feeling among us Ada zealots was that the DeLauer memo was premature and actually worked against the cause of Ada. Because of the lack of tools, compilers, and trained programmers, many developers either received a waiver from the Ada mandate or simply ignored the memo. Sort of like the Maginot line – folks just went around it.

However, time has been good to Ada. It has been updated several times, and the actual intent of the DeLauer memo (that high-level languages be used to develop DoD software) has long since been met. Back in the 1980s, as I said, there were literally hundreds of languages being used. Today, most software is created using a relative few languages. C++ (nobody uses C anymore) and Java are probably most used, and according to trends, C++ usage is going down while Java is on the rise; Java provides almost all of the same safety features (strong typing, data abstraction, encapsulation). There are quite of few us who hold the opinion that Ada strongly influenced Java – and that Java has C++ syntax, but Ada semantics. Ada is still widely used outside of the United States, and Ada is used worldwide in the avionics industry.

Ada is still a viable force in avionics simply because it's very good at what it was designed to do – provide high-quality code in safety-critical environments. It has run-time features such as real-time and parallel processing that are hard to find in any other language.

It's all about safety and security – the same things that the Maginot Line was designed to give. And, just like the Maginot Line, it all lies in the execution.

— David A. Cook, Ph.D.
The AEgis Technologies Group, Inc.
<dcook@aegistg.com>

Additional Reading

1. Much of this research comes from <http://europeanhistory.about.com/library/weekly/aa070601a.htm> and <http://en.wikipedia.org/wiki/Maginot_Line>.
2. See <www.people.ku.edu/~nkinners/LangList/Extras/langlist.htm> for a list of more than 3,000 languages.
3. See <http://oop.rosweb.ru/> under Language List, then Ada.
4. See "Evolutionary Trends of Programming Languages." This excellent article can be found at <www.stsc.hill.af.mil/crosstalk/2003/02/schorsch.html>.
5. "An Empirical Study of Programming Language Trends," Dios et. al., IEEE Software, May/June 2005.

BUILDING SOLUTIONS FOR THE SYSTEMS OF THE PAST, PRESENT, AND FUTURE!

If you are tired of spending more and getting less, let us—a successful organization with a proven track record—help you.

We are customer- and user-oriented, dedicated to providing low-cost solutions and high-quality products.

OGDEN AIR LOGISTICS CENTER

WE CAN SUPPORT YOUR DEVELOPMENT AN[illegible]AINMENT NEEDS:

Softwar[illegible] Engineeri[illegible] | [illegible]ste[illegible] [illegible]neering | [illegible]b-Bas[illegible] [illegible]esign | [illegible]oftware Configuration Management

Hardware Engineering | [illegible]echnol[illegible] | Simulation and Emulation | Consulting Services

ON A WIDE RANGE [illegible] SYSTEMS AND PRO[illegible]UCTS...

- Avionics
- Automatic Test Equipment
- Electronics
- C^3I
- Weapons
- Mission Planning
- Web-Based Products
- Space and Missile Systems
- Ground Support Equipment

...AND MANY MORE

PLEASE CONTACT US TODAY

Ogden Air Logistics Center
309th Software Maintenance Group
(Formerly MAS Software Engineering Division)
Hill Air Force Base, Utah 84056

Commercial: (801) 777-2615
DSN: 777-2615
E-mail: ooalc.masinfo@hill.af.mil
or visit our Web site:
www.mas.hill.af.mil

CrossTalk is co-sponsored by the following organizations:

76th Software Maintenance Group

309th Software Maintenance Group

402nd Software Maintenance Group

Homeland Security

NAVAIR

CrossTalk/517 SMXS/MDEA
6022 Fir AVE
BLDG 1238
Hill AFB, UT 84056-5820

PRSRT STD
U.S. POSTAGE PAID
Albuquerque, NM
Permit 737