# Safety and Security: Certification Issues and Technologies[1]

Dr. Benjamin M. Brosgol
*AdaCore*

*Many military systems are safety-critical, with failure possibly resulting in loss of human life. In today's interconnected environment, safety requires security. Compliance with the higher levels of safety or security standards demands a disciplined development process along with appropriate programming language and toolset technology. Since full, general-purpose languages are too large and complex to be usable for safety-critical or high-security systems, a key requirement is to define subsets that are simple enough to facilitate certification but expressive enough to program the needed application functionality. This article summarizes representative safety and security standards (DO-178B and the Common Criteria, respectively), identifies the language-related issues surrounding safety and security certification, and assesses three candidate technologies—C (including C++), Ada (including SPARK), and Java—with respect to suitability for safety-critical or high-security systems.*

Compliance with formal safety standards is becoming a major consideration, and sometimes a requirement, in defense systems. The safety standard that is most relevant is DO-178B [1], which is used by the Federal Aviation Administration (FAA) for the certification of commercial aircraft. DO-178B is process-oriented; certification is not so much a safety assessment of the completed system, but rather an evaluation of a set of developer-provided artifacts. These artifacts are intended to provide both direct evidence of sound software engineering practice and lend assurance (through indirect evidence) of the safety of the resulting system.

DO-178B identifies five levels of criticality: A (highest) through E (lowest). Table 1 characterizes the various levels and identifies the number of DO-178B requirements that apply. The term *with independence* means that the evidence for meeting a requirement must be supplied by someone other than the developer. The term *safety-critical* generally applies to software at levels A and B, which demand greater rigor and more comprehensive analysis than the lower levels.

DO-178B focuses on requirements-based testing and bi-directional traceability (from requirements to code, and vice versa) as key elements of software verification. Test cases must fully cover the code; *dead code* (unexercised code that does not correspond to a specific requirement) must be removed.

DO-178B is open to criticism on several grounds:

- It does not directly assess the safety of the resulting system.
- Although the artifacts are process-oriented, there is no guarantee that sound processes were followed and it is not rare for developers to prepare the DO-178B artifacts for previously developed components *a posteriori*.
- Its emphasis on testing does not adequately take into account alternative technologies (such as formal methods) for providing safety assurance.
- It is unclear on how its objectives apply to modern software development approaches such as object-oriented technology (OOT).

These problems should not be overemphasized. DO-178B has been successful in practice: Although there have been some close calls, DO-178B-certified software has never been the direct cause of an aircraft accident resulting in a fatality. However, much has changed in the software industry since the early 1990s when DO-178B was written. Work is in progress on a successor, DO-178C [2], that will attempt to address some of the perceived issues with DO-178B. For example, DO-178C will accommodate newer software technologies (OOT, model-based design) and alternative software verification techniques (formal methods, abstract interpretation).

## Security Certification

Security is generally defined as the protection of assets against threats to their *confidentiality*, *integrity*, and/or *availability*. Designing an information technology (IT) product for security thus involves design steps (avoiding vulnerabilities that adversaries could exploit to compromise these requirements) as well as runtime actions (detecting and responding to attempted breaches).

DO-178B says nothing explicit about security, but a system with security vulnerabilities is at risk for exploitation by adversaries to render it unsafe. The *safety requires security* principle has two implications. First, an organization developing safety-critical software should adopt methodologies and design frameworks that can help realize security requirements. Guidelines such as the "Defense Acquisition Guidebook" [3] and architectures such as multiple independent levels of security (MILS) [4] are relevant. Second, it must be possible to assess whether a product with safety-critical requirements is sufficiently secure. The Common Criteria/Common Evaluation Methodology [5] provides such an assessment mechanism. These international standards include a catalog of security-functional and assurance requirements and a process for evaluating the security characteristics of a given IT product.

Somewhat analogous to the levels of DO-178B, the Common Criteria defines seven Evaluation Assurance Levels (EALs), numbered from 1 to 7 in increasing order of criticality. Generally speaking, EAL 4 corresponds to best commercial practice without a serious focus on security threats; higher levels require special security-oriented mechanisms and increased

Table 1: *Criticality Levels in DO-178B*

| Level | Condition | Effect of Anomalous Behavior | Number of Objectives |
|---|---|---|---|
| A | Catastrophic failure | "... prevent continued safe flight and landing..." | 66 (14 *with independence*) |
| B | Hazardous/severe failure | "... serious or potentially fatal injuries to a small number of … occupants ..." | 65 (11 *with independence)* |
| C | Major failure | "... discomfort to occupants, possibly including injury..." | 57 |
| D | Minor failure | "... some inconvenience to occupants..." | 28 |
| E | None | "... no effect on aircraft operational capability or pilot workload..." | 0 |

effort in demonstrating compliance. At EAL 7, formal methods (i.e., mathematics-based analyses) are required to demonstrate that security requirements are met.

With safety implying the need for security, it is reasonable to consider certifying a system against standards for both. This idea is not new; the SafSec project [6], sponsored by the United Kingdom's Ministry of Defense, has developed an integrated methodology for dual safety and security certification for avionics. In another effort, a group at the University of Idaho has analyzed the correspondence between DO-178B and the Common Criteria, mapping DO-178B objectives to Common Criteria elements [7], and has studied the feasibility of joint certification. However, the practicality of applying the Common Criteria to large Department of Defense (DoD) systems is unclear. As summarized in a 2007 Report of the Defense Science Board Task Force:

> Criticisms of Common Criteria-based schemes are that they are expensive, require artifacts that are not produced until well after product design and implementation, do not substantially reduce implementation-level vulnerabilities when using today's software development practices, and lack thorough penetration analysis at EAL 4 and below. [8]

Furthermore, the fact that a product has been certified at a specific EAL means very little by itself. First, it says nothing about the quality of the product outside the security functional requirements. Second, a prospective consumer needs to understand which of these requirements are being implemented and whether the vendor-assumed operational environment (the severity of the threats/assumed skills of the adversaries) matches reality.

Notwithstanding how it is assessed, security is obviously necessary for safety, and safety certification agencies are paying increasing attention to the relationship between the two. As an example, an FAA "Special Conditions" notice directed one supplier to demonstrate the independence of the networks for passenger-accessible components and flight control on one of its aircraft [9].

## A Comparison of Safety and Security Certification Issues

DO-178B and the Common Criteria have some basic similarities:

- Concern with the full software development life cycle—including peripheral activities such as configuration management—in an attempt to catch human (developer) error before the system is fielded.
- Tiered approach (criticality levels) reflecting real-life trade-offs: Resources are finite, and a system must be safe/secure enough for its intended purpose.
- Emphasis on testing as a major element of software verification.

There are also some important differences:

- **Scope of requirements.** DO-178B deals with the entire system; the Common Criteria focuses almost exclusively on just the security functional requirements.
- **Functional requirements.** There is no specific set of safety functions called out in DO-178B. In contrast, the security domain has well-defined functional requirements that need to be implemented.
- **System users/operators.** An IT product must be immune to attacks from unknown and possibly malevolent users who can directly supply input. Input to a safety-critical system is generally supplied by known operators whose trustworthiness has been separately vetted.

In one sense, compliance with safety standards is more demanding:

- Each component must be certified against requirements for its safety level.
- At the higher levels, it is necessary to both demonstrate the absence of dead code and perform structural testing to verify the absence of non-required functionality.
- For EAL compliance, the specific development and testing requirements apply only to the security functions and not to the entire IT product; there is no prohibition against dead code/extra functionality (although such code must be shown to be free from vulnerabilities).

In another sense, compliance with security standards is more demanding:

- Formal methods are required at EAL 7.
- Vulnerability analysis is difficult and must assume a sophisticated and malevolent adversary; for safety, the adversaries are the laws of physics.

Although safety requires security, the relationship in the other direction is not so immediate. Most IT products for which security is critical do not control systems where life is at stake and, thus, safety is generally not an issue.

In the context of overall system design, safety and security sometimes conflict, especially with respect to behavior under failure conditions. Taking a system offline to protect data may be reasonable behavior for security, but if the data are needed for flight control/management, then such a policy may have disastrous consequences for safety. *Fail-safe* is not the same thing as *fail-secure*. These sorts of conflicts need to be resolved during design, with appropriate trade-offs based on the anticipated risks.

## Programming Language Requirements

The programming language choice is arguably the most important technical decision that the developer organization will make. As summarized in a National Academy of Sciences report:

> The overwhelming majority of security vulnerabilities reported in software products—and exploited to attack the users of such products—are at the implementation level. The prevalence of code-related problems, however, is a direct consequence of higher-level decisions to use programming languages, design methods, and libraries that admit these problems. [10]

Although directed at security issues, these comments apply equally to safety. The programming language plays a key role in determining the ease or difficulty of developing software that avoids vulnerabilities and that is certifiable against safety or security standards.

A simple example of a programming language feature that can easily lead to an application vulnerability is the C library function *gets()*, which reads a character string as input from a user until an *end-of-line* or *end-of-file* is encountered. The program can only pre-allocate an area of some fixed length as the destination, but a user can accidentally or intentionally supply input that exceeds this bound. The effect is the classical *buffer overflow*, in which the excess characters overwrite other data, possibly including a function's return address. By crafting an input string with specific content, a malevolent user can take control of the machine to execute arbitrary code.

Although DO-178B and the Common Criteria do not offer direct guidance on a programming language choice, it is possible to abstract from their specific objectives and infer several general requirements that a language must meet. The following sections discuss four of these requirements: reliability, predictability, analyzability, and expressiveness.

### *Reliability*

The language should promote the development of readable, correct code as well as:

- Have an intuitive lexical and syntactic structure and be free of *traps* and *pitfalls*.
- Help detect errors early (at compile time if possible), and should prevent errors such as out-of-range array indices and references to uninitialized data.
- Help (if it supports concurrent programming with *threads/tasks*) in avoiding errors such as unprotected accesses to shared data, race conditions (where the effect of the program depends on the relative speed of the threads/tasks), and deadlock.

### *Predictability*

The language specification should be unambiguous. Implementation-dependent or, worse, undefined behavior introduces vulnerabilities because the effect of the program may not be as the developer had intended.

### *Analyzability*

The language should facilitate both static analysis (detecting uninitialized variables, identifying dead code, predicting maximum stack usage and worst-case execution time, etc.) and dynamic analysis (requirements-based or structure-based testing). A useful catalog of such analysis techniques (from [11]) is given in Table 2.

The various analysis techniques impose constraints on the programming language. For example, control and data flow analyses generally prohibit the use of *goto* statements, stack usage analysis generally prohibits recursion, and coverage analysis may preclude the use of source code constructs that generate implicit loops or conditionals. As a result, there is no such thing as *the* safety-critical or high-security subset of a given language. The particular subset used either determines or is determined by the analysis techniques that will assist in demonstrating compliance with the operative certification standard.

Automated static analysis tools play an important role in the safety and security domains; indeed, there is a U.S. Department of Homeland Security-sponsored project under way—Static Analysis Metrics and Tool Evaluation (SAMATE) [12]—identifying and measuring the effectiveness of such tools. Static analysis tools are most successful during program development, where they can help detect problems before they occur, versus as a technique for detecting vulnerabilities *a posteriori* in existing code.

| Approach | Group Name | Technique(s) |
|---|---|---|
| *Static Analysis* | Flow Analysis | Control Flow |
| | | Data Flow |
| | | Information Flow |
| | Symbolic Analysis | Symbolic Execution |
| | | Formal Code Verification |
| | Range Checking | Range Checking |
| | Stack Usage | Stack Usage |
| | Timing Analysis | Timing Analysis |
| | Other Memory Usage | Other Memory Usage |
| | Object Code Analysis | Object Code Analysis |
| *Dynamic Analysis (Testing)* | Requirements-Based Testing | Equivalence Class |
| | | Boundary Value |
| | Structure-Based Testing | Statement Coverage |
| | | Branch Coverage |
| | | Modified Condition/ Decision Coverage |

Table 2: *Analysis Techniques*

### *Expressiveness*

The language should support general-purpose programming, either through language features or auxiliary libraries, and should also offer specialized functionality (as required). For real-time safety-critical systems, this means support for interrupt handling, low-level programming, concurrency, perhaps fixed-point arithmetic, and other features. For high-security systems, it means mechanisms are needed for implementing security-functional requirements (e.g., for cryptography).

Unfortunately, language generality (as implied by the expressiveness requirement) directly conflicts with the analyzability requirement. That conflict complicates the language selection decision.

## Object-Oriented Technology

The history of programming languages has seen a steady evolution of features that promote maintainable software and many of these features directly support the reliability and analyzability requirements previously described. However, some of the advances present difficulties for safety and security certification. Perhaps the most significant example is OOT [13], found in such languages as C++, Ada 95, and Java. OOT is not addressed in DO-178B, and there are indeed a number of challenges:

- **A paradigm clash.** OOT's distribution of functionality across classes conflicts with DO-178B's focus on tracing between requirements and implemented functions.
- **Technical issues.** The features that are the essence of OOT complicate safety and security certification. For example, dynamic binding typically is implemented by a compiler-generated data structure known as a *vtable* (a table of addresses of functions). For safety or security certification, the developer must demonstrate that the *vtable* is properly initialized and that it cannot be corrupted.
- **Cultural issues.** Certification authority personnel are not necessarily language experts and may (rightfully) be concerned about how to deal with unfamiliar technology.

A series of workshops several years ago produced a handbook [14] that addressed these issues in detail. The in-progress work on DO-178C is taking these into account, and it is likely that the eventual new standard will offer some direct guidance in connection with OOT. However, developers are not waiting for DO-178C. OOT is currently being used in safety-critical code; as one example, an avionics system using Ada 95's object-oriented features has been certified at Level A. It seems inevitable—as experience with OOT and certification is gained—that usage of object-oriented languages will increase.

## Candidate Programming Languages

Although (in principle) any programming language could be used for developing safety-critical or high-security software, the requirements for reliability, predictability, and especially analyzability imply that suitable subsets be chosen. The key issues are how a language can be subsetted to ease certification for applications restricted to the subset, and whether the language has intrinsic problems that cannot be removed by subsetting.

This section summarizes how several current language technologies—either

currently in use or under consideration for safety-critical systems—compare with respect to subsettability.

### *C-Based Technology*[2]

#### MISRA-C

The United Kingdom-based Motor Industry Software Reliability Association (MISRA) has produced a set of language restrictions, called MISRA-C [16], which attempts to mitigate C's vulnerabilities. MISRA-C codifies best practices for C programming and has become somewhat of a *de facto* standard as a C subset for critical systems. Benefits stem from C's relative simplicity, the large population of C programmers, and a wide assortment of tools and service providers. MISRA-C has been used successfully in safety-critical systems.

On the other hand, MISRA-C has some significant drawbacks:

- C was not designed for safety-critical systems, and some intrinsic issues (e.g., the wraparound semantics for integer overflow) cannot be removed by subsetting.
- Despite MISRA-C's stated goals, the rules are not always enforceable by static tools, and different tools may enforce the subset differently.
- Since concurrency is not provided in C (it is only available through external libraries), MISRA-C offers no guidance on how to use C in a multithreaded environment.

#### C++

C++ [17] is in many ways a better C. Developers of safety-critical systems often have staff knowledgeable in C++ and may possess existing C++ components that they would like to re-use in a certified system.

To help meet this goal, coding standards such as Joint Strike Fighter C++ [18], and MISRA C++ [19] have been developed. These extend or adapt MISRA-C to deal with C++'s additional facilities. The rules constrain the usage of language features in order to avoid problems and to promote good style.

Safe C++ coding standards are essential if C++ is chosen, and C++ has been used to develop safety-critical systems. However, the previously noted drawbacks for MISRA-C apply here, and the OOT coding guidelines do not solve the underlying certification issues.

### *Ada-Based Technology*

#### Ada

Ada [20] was designed to be used for safety-critical systems. It avoids many of the C and C++ vulnerabilities (e.g., checking for out-of-range array indexing and integer overflow), and also offers a standard set of concurrent programming features. Ada continues being widely used for safety-critical systems including military and commercial avionics.

Full Ada is too large to be practical for safety certification so subsetting is required. Ada provides a unique approach to this issue, allowing the application to specify the features that are to be excluded. This means no runtime support libraries for such features, and compile-time error detection of attempted uses. The *à la carte* style to defining language subsets is flexible and does not require specialized tool support: a standard compiler performs the necessary analysis.

The latest Ada language standard also includes the Ravenscar profile [21], a certifiable subset of concurrency features.

> *"Ada continues being widely used for safety-critical systems including military and commercial avionics."*

Ada's disadvantages for safety-critical systems are largely external (non-technical). Ada usage is not as widespread as other languages and, thus, its tool vendor community is smaller. On the technical side, Ada does not directly address vulnerabilities such as references to uninitialized variables. As with C and C++, supplementary analysis is required to detect/prevent such errors.

#### SPARK

SPARK [22] is a subset of Ada 95, augmented by specially formed comments known as *contracts* (or *annotations*), designed to facilitate a rigorous, static demonstration of program correctness. SPARK omits features that complicate analysis or formal proofs or that interfere with bounded time/space predictability. The language includes most of Ada 95's static features as well as the Ravenscar concurrency profile, and the semantics are completely unambiguous (no implementation-dependent or undefined behavior).

Contracts in SPARK specify data and information flow, inter-module dependencies, and dynamic invariants (pre-/post-conditions, assertions). The SPARK tools analyze the program to ensure that the code is consistent with the contracts and that no runtime exceptions will be raised. They detect errors such as potential references to uninitialized variables and dead code. The static analysis performed by the SPARK tools is *sound* (there are no *false negatives*, an especially important requirement in connection with safety certification) with a low false alarm rate (there are few *false positives*). The SPARK tools can also generate verification conditions and automate the proof of these conditions.

SPARK has been used in practice on a variety of systems, both safety-critical and high-security. Of all the candidate language technologies, SPARK best meets the requirements for reliability, predictability, and analyzability. Its main technical drawback is with expressibility, as it has a rather restricted feature set. Additionally, the SPARK infrastructure (user/vendor community) is smaller than that of other language technologies.

### *Java-Based Technology*

Java [23] seems simultaneously logical and curious as a technology choice for safety-critical systems. On one hand, it was designed with careful attention to security: Its initial goal was to enable downloadable *applets* to be executed on client machines without risk of compromising the confidentiality or integrity of client resources. The Java language is largely free from the implementation dependencies found in C, C++, and Ada, such as order of expression evaluation. Java also performs conservative checks to prevent unreachable (dead) code and references to uninitialized variables. It provides automatic storage management (*garbage collection*) instead of an explicit *free* construct that is the source of subtle errors in other languages.

Security, however, is not the same as safety. Indeed, Java technology has limitations for safety-critical systems, falling into two general categories:

1. Ensuring real-time predictability.
2. Meeting certification standards such as DO-178B.

Both of these have been the subject of Java Specification Requests (JSRs) under Sun Microsystems' Java Community Process [24]. JSRs 1 [25] and 282 [26] have defined the Real-Time Specification for Java (RTSJ); JSR-302 [27], in progress, is defining a subset of the RTSJ that is intended for Java applications that need to be certified to DO-178B at levels up to A.

The RTSJ extends the Java platform to add real-time predictability. The main enhancements are for concurrency (to define scheduling semantics more precisely than in standard Java, and to prevent pri-

ority anomalies) and memory management (to avoid garbage collection interference).

The RTSJ, as an extension of the standard Java platform, is not appropriate (and was not intended) for safety-critical applications. It is too complex and some of its major features (especially in the memory management area) require runtime checks that may be expensive. However, it does address Java's real-time issues and, thus, is serving as the basis for JSR 302's safety-critical Java specification. This in-progress effort defines three levels of support for safety-critical systems: (1) a traditional cyclic executive (no threading); (2) a thread-based approach with simple memory management; and (3) a thread-based approach with more general memory management. Each is characterized by a corresponding subset of RTSJ functionality and Java class libraries. JSR-302 exploits Java 1.5's annotation feature: A developer annotates various properties of the code (for example, memory usage), and static analysis tools verify the annotations' correctness.

Of all the language technologies that are candidates for safety-critical development, Java has the most significant challenges:

- The Virtual Machine execution environment for Java programs is unconventional, blurring the distinction between code and data and raising safety certification issues.
- Unlike C++ and Ada (where OOT is available but optional), Java is based around object orientation. It is possible to use Java without taking advantage of OOT, but the style is contrived.
- Java is lexically and syntactically based on C, therefore sharing a number of that language's traps and pitfalls.
- The RTSJ/JSR-302 approach to memory management is rather complicated, and requires Java programmers to carefully analyze dynamic memory usage.

Despite these issues, there is interest in safety-critical Java from both the developer and the user communities. An organization that has adopted Java as an implementation language on a project may have some components with safety-critical requirements, and keeping the entire system within one language can simplify some aspects of project management.

## Conclusion

Developing safety-critical/high-security systems is difficult. The key skill is not so much the knowledge of a particular programming language; a software professional should be able to learn a new language in a short amount of time. The more critical talent is the ability to think through a design and implementation with a focus not just on meeting a system's functional requirements but also on avoiding hazards and vulnerabilities. Such *negative programming*—ensuring that bad things do not happen—requires careful analysis and a defensive development approach that, in turn, places demands on the programming language and tools. For software safety and security, the idea of *minding your language* is more than a matter of etiquette; it could be the key to a system's success.◆

## References


1. RTCA SC-167/EUROCAE WG-12. RTCA/DO-178B. Software Considerations in Airborne Systems and Equipment Certification. Dec. 1992.
2. Embry-Riddle Aeronautical University. "Software Considerations in Airborne Systems." <http://forum.pr.erau.edu/SCAS>.
3. Defense Acquisition University. Defense Acquisition Guidebook. 20 Dec. 2004 <http://akss.dau.mil/dag/>.
4. Vanfleet, W. Mark, et al. "MILS: Architecture for High-Assurance Embedded Computing." CrossTalk Aug. 2005 <www.stsc.hill.af.mil/crosstalk/2005/08/0508Vanfleet_etal.html>.
5. The Common Criteria Portal. Common Criteria for Information Technology Security Evaluation. Vers. 3.1. Sept. 2006 <www.commoncriteriaportal.org/thecc.html>.
6. United Kingdom Ministry of Defense and Praxis High Integrity Systems. SafSec Methodology: Standard. 2 Nov. 2006 <www.praxis-cs.com/safsec/downloads/SafSec_Methodology_Standard_Material_pdf.pdf>.
7. Taylor, Carol, Jim Alves-Foss, and Bob Rinker. "Merging Safety and Assurance: The Process of Dual Certification for Software." STC 2002 <www.csds.uidaho.edu/papers/Taylor02d.pdf>.
8. Defense Science Board. Report of the Defense Science Board Task Force on Mission Impact of Foreign Influence on DoD Software. Washington: Office of the Under Secretary of Defense for Acquisition, Technology, and Logistics. Sept. 2007: 33.
9. Department of Transportation, FAA. "Special Conditions: Boeing Model 787-8 Airplane; Systems and Data Networks Security – Isolation or Protection From Unauthorized Passenger Domain Systems Access." Federal Register. 2 Jan. 2008.
10. Jackson, Daniel, Martyn Thomas, and Lynette I. Millett, Eds. Software for Dependable Systems: Sufficient Evidence? Washington: National Academies Press. 27 Aug. 2007 <http://books.nap.edu/catalog.php?record_id=11923>.
11. International Organization for Standardization. ISO/IEC TR 15942, Guide for the Use of Ada in High-Integrity Systems. 2000 <http://std.dkuug.dk/JTC1/SC22/WG9/n359.pdf>.
12. National Institute of Standards and Technology. SAMATE – Software Assurance Metrics and Tool Evaluation. July 2005 <http://samate.nist.gov>.
13. Meyer, Bertrand. Object-Oriented Software Construction. 2nd ed. Santa Barbara, CA: Prentice Hall PTR, 1997.
14. FAA. Handbook for Object-Oriented Technology in Aviation. 26 Oct. 2004 <www.faa.gov/aircraft/air_cert/design_approvals/air_software/oot>.
15. International Organization for Standardization. ISO/IEC 9899: 1990; Programming Languages - C.
16. MISRA-C: 2004. Guidelines For the Use of the C Language in Critical Systems. Nuneaton, U.K.: MISRA Ltd., 2004.
17. International Organization for Standardization. ISO/IEC 14882: 2003; Programming Languages – C++.
18. Lockheed Martin Corp. Joint Strike Fighter Air Vehicle C++ Coding Standards for the System Development and Demonstration Program. Dec. 2005 <www.jsf.mil/downloads/documents/JSF_AV_C%2B%2B_Coding_Standards_Rev_C.doc>.
19. MISRA-C++: 2008. Guidelines For the Use of the C Language in Critical Systems. Nuneaton, U.K.: MISRA Ltd., 2004.
20. International Organization for Standardization. ISO/IEC 8652:1995/Amd 1: 2007 Programming Languages - Ada.
21. Dobbing, Brian, and Alan Burns. "The Ravenscar Profile for Real-Time and High Integrity Systems." CrossTalk Nov. 2003 <www.stsc.hill.af.mil/crosstalk/2003/11/0311Dobbing.html>.
22. Barnes, John. High Integrity Software – The SPARK Approach to Safety and Security. Addison-Wesley, 2003.
23. Arnold, Ken, James Gosling, and David Holmes. The Java Programming Language. 4th ed. Addison Wesley, 2006.
24. Community Development of Java Technology Specifications. The Java Community Process℠ Program. 2008 <www.jcp.org/en/home/index>.

## Coming Events

**December 1-3**
*Defense Network Centric Operations*
Arlington, VA
www.wbresearch.com/DNCO

**December 1-4**
*26th Army Science Conference*
Orlando, FL
www.asc2008.com

**December 1-4**
*Interservice/Industry Training, Simulation, and Education Conference*
Orlando, FL
www.iitsec.org

**December 7-10**
*Winter Simulation Conference*
Miami, FL
http://wintersim.org

**November 16-18**
*Software Engineering and Applications*
Orlando, FL
www.iasted.org/conferences/home-632.html

**November 18-20**
*SpecOps East 2008*
Fayetteville, NC
www.defensetradeshows.com/SPECOPSEAST08_General_Info.html

**November 18-22**
*Joint International Conference on Engineering and Technology*
Nashville, TN
www.ijme.us/IJME_Conference_2008/

**April 20-23, 2009**
*21st Annual Systems and Software Technology Conference*
Salt Lake City, UT
www.sstc-online.org

**Coming Events: Please submit coming events that are of interest to our readers at least 90 days before registration. E-mail announcements to: nicole.kentta@hill.af.mil.**


25. Community Development of Java Technology Specifications. JSR 1: RTSJ. 12 July 2006 <www.jcp.org/en/jsr/detail?id=1>.
26. Community Development of Java Technology Specifications. JSR 282: RTSJ. 12 Sept. 2005 <www.jcp.org/en/jsr/detail?id=282>.
27. Community Development of Java Technology Specifications. JSR 302: Safety Critical Java™ Technology. 24 July 2006 <www.jcp.org/en/jsr/detail?id=_302>.


## Notes

1. This article is based on a tutorial, "Safety and Security: An Analysis of Certification Issues and Technologies," presented by the author at the Systems and Software Technology Conference, 2008.
2. In this section, C means the 1990 version of the ISO language standard [15].

## About the Author


**Benjamin M. Brosgol, Ph.D.,** is a senior member of the technical staff at AdaCore. He has been actively involved with the Ada language effort since its outset and has received the SIGAda Outstanding Ada Community Contributions award. Brosgol has a bachelor's degree in mathematics from Amherst College, as well as master's and doctorate degrees in applied mathematics from Harvard University.

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


## Letter to the Editor

**Dear CrossTalk Editor,**

Reading August's 20th anniversary edition—especially Gary Petersen's CrossTalk*: The Long and Winding Road*—triggered my own early memories of the journal and the Software Technical Support Center (STSC). I have always appreciated CrossTalk, from the authors' real-world application of concepts to the "above-and-beyond" assistance of your staff. We at Northrop Grumman put what we learned into practice.

In 1989, our group at Northrop (before adding the Grumman) utilized your articles on the Department of Defense's (DoD) requirements for the Capability Maturity Model Integration (CMMI®) Level 3. While I cannot remember the authors or titles, these articles aided our logistics engineers in developing a quality approach as we started our transformation into what is now much-renowned Software Engineering Processing Group.

In the mid-90s, Watts S. Humphrey—who was, not surprisingly, part of the 20th anniversary issue—was one of several CrossTalk authors whose articles pointed the way toward DoD systems management of large-scale software and systems engineering integration. As well, articles on software project management were one of the tools used to kick off Northrop's CMMI Level 3 effort and organize a more integrated project management approach to B-2 software engineering.

Around this same time, the editorial team of CrossTalk invited Northrop personnel to participate in STSC meetings and presentations, and introduced us to senior DoD system managers. These sessions were the genesis of our software engineering and systems engineering breakthroughs. And, although it may seem like a small gesture, supplying us with the proceedings from these gatherings helped educate and motivate our teams of software engineers and supported our argument that we needed new hardware and better computer-aided software engineering tools.

We saved more than 100,000 man-hours by implementing methodologies gleaned from CrossTalk, equating to approximately $8.3 million per year (in 1993 dollars) or $33 million over the four-year period of development. This is quite a savings when compared to our estimated $121 million annual budget. We've all received plentiful kudos for our achievements, but we would like to pass along some of that gratitude to CrossTalk's authors and staff.


—John B. Burger
Northrop Grumman (retired)
4940 Flora Vista LN
Sacramento, CA 95822
<jjburger@aol.com>


® CMMI is registered in the U.S. Patent and Trademark Office by Carnegie Mellon University.