



Designing High-Security Systems: A Comparison of Programming Languages

SSTC 2007
Tampa, FL

June 19, 2007

US Headquarters:

104 Fifth Avenue, 15th Floor
New York, NY 10011

+1-212-620-7300 (voice)
+1-212-807-0162 (FAX)

www.adacore.com

Ben Brosgol • brosgol@adacore.com
Greg Gicca • gicca@adacore.com

AdaCore **Outline**

Security-critical systems

- Common Criteria / Common Evaluation Methodology
- Implications for programming languages
- Comparison with requirements for safety-critical systems
- Challenges from modern programming language features
 - Object-Oriented Technology
 - The need for subsets

Assessment of language approaches

- C-based languages
 - MISRA C, C++ subset
- Ada-based languages
 - Ada subset, SPARK
- Java-based languages
 - Java, Real-Time Specification for Java, Safety-Critical Java Technology

Conclusions

1

AdaCore Common Criteria / Common Evaluation Methodology

International standards for Information Technology (IT) security

- ISO/IEC 15408, Parts 1-3: Criteria for IT Security Evaluation
 - Part 1: Introduction and general model
 - Part 2: Security functional requirements
 - Part 3: Security assurance requirements
- ISO/IEC 18045: Common Methodology for IT Security Evaluation

Purpose

- "... provide a complete methodology for specifying IT security requirements, designing a solution to meet those requirements, and conducting an independent evaluation to ensure that all security requirements have been implemented ... correctly." [Herrmann, p. 54]
- Usable by customers, developers, evaluators

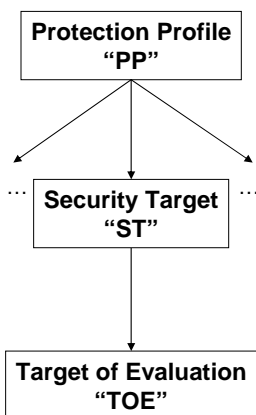
Security concern

- Protect IT assets from threats that may compromise confidentiality, integrity, and/or availability

[Herrmann] Debra S. Herrmann, *Using the Common Criteria for IT Security Evaluation*, Auerbach Publications, 2003.

2

AdaCore Common Criteria Artifacts



Consumer-specified implementation-independent specification of security requirements:

- Security functional requirements (SFRs)
- Security assurance requirements (SARs) ⇔ Evaluation Assurance Level (EAL)

Vendor-supplied implementation-dependent design based on a PP

- Specifies security mechanisms to meet PP requirements
- Basis for developing a Target of Evaluation
- One PP may generate many STs

Vendor-supplied implementation of a ST

- "IT product, system of network and its associated administrator and user guidance documentation that is the subject of an evaluation" [Herrmann, p. 14]
- What gets evaluated are principally the TOE's Security Functions (TSFs), which are the implementation of the Security Functional Requirements specified in the PP

3

Level	Description	Assurance
EAL 1	Functionally tested	Low
EAL 2	Structurally tested	
EAL 3	Methodically tested and checked	
EAL 4	Methodically designed, tested, and reviewed	Medium
EAL 5	Semiformally designed and tested	
EAL 6	Semiformally verified design and tested	
EAL 7	Formally verified design and tested	Highest

Each EAL implies a specific set of Security Assurance Requirements

- Most SAR categories apply only to the implementation of the TOE's Security Functional Requirements (e.g. Development, Testing families)
- Some SARs apply to the whole TOE (e.g. Vulnerability Assessment)

4

Reliability

- Support for development of readable, correct code
 - No "traps and pitfalls"
 - Intuitive lexical and syntactic features
- Avoidance of references to uninitialized data, concurrency bugs, etc.
- Early error detection
 - Compile-time checking whenever possible
 - Run-time checking otherwise (e.g. array index in range)

Predictability

- Unambiguous language semantics
 - No undefined / implementation-dependent behavior
- Ability to demonstrate statically that time and space constraints will be satisfied
 - Deadlines will be met
 - Stack and heap space will not be exhausted

5

AdaCore Language Requirements Implied by CC/CEM (2)

Analyzability

- Control and data flow analysis
 - Detects / prevents references to uninitialized data
- Information flow analysis
 - Identify dependencies between a routine's inputs and outputs
- Formal code verification
- Range checking
- Traceability between program representations and different levels
- Dynamic analysis (testing)

Expressiveness

- General-purpose features consistent with the above requirements
- Functionality for specialized processing, e.g. for embedded systems
 - Interrupt handling
 - Low-level programming
 - Fixed-point arithmetic

6

AdaCore Comparison of Security and Safety Issues

Safety-critical systems need to meet standards such as DO-178B

- Levels E (lowest) through A (highest) relate to the potential effect of a failure
- For each level there is a set of requirements that must be met

Compliance with safety standards is in some sense more demanding

- Each component must be certified against the requirements applicable for that component's safety level
- At the higher levels it is necessary to 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 TSFs and not to the entire TOE

- No prohibition against dead code / extra functionality
- But vulnerability analysis needs to consider the entire TOE

The difference is in part because security functional requirements often correspond to specific functions in the code

- Safety is more an issue that is reflected in how components are designed

Despite these differences, the language issues are similar

7

AdaCore Verification Techniques*		
Approach	Group Name	Technique
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

* From ISO/IEC TR 15942:2000(E), *Guide for the use of the Ada programming language in high integrity systems* 8

AdaCore Modern Language Features and CC/CEM
<p>Many features help</p> <ul style="list-style-type: none"> • Encapsulation / namespace control ⇒ less data coupling ⇒ analyzability • Strong typing, compile-time checking ⇒ early error detection <p>Some features present difficulties</p> <ul style="list-style-type: none"> • Need for expressive power in general-purpose language, versus need to constrain feature usage <ul style="list-style-type: none"> ▪ Demonstrate correctness of TSFs at higher EALs ▪ Demonstrate that TOE's non-TSF parts do not introduce vulnerabilities • Desire for dynamic flexibility, versus need for static analysis • Desire for high-level features (language constructs match problem space), versus need to certify the code that actually runs <ul style="list-style-type: none"> ▪ Run-time library, API need particular attention ▪ How, if at all, use features such as exceptions, concurrency, dynamic allocation • Desire that compiler generate efficient code / exploit machine or OS capabilities, versus desire for implementation-independent program semantics

9

AdaCore Object-Oriented Technology (“OOT”)

What is OOT?

- Software development methodology supported by language features
 - Primary focus is on data elements and their relationships
 - Secondary focus is on the processing that is performed
- Applicable during entire software “life cycle”

Language concepts (OOP, or “Object-Oriented Programming”)

- *Object* = state (“attributes”) + operations (“methods”)
 - *Class* = module + object creation template
 - *Encapsulation* = separation of interface (spec for methods) from implementation (state, algorithms)
 - *Inheritance* = specialization (“is-a”) relationship between classes
 - Extend a class, adding new state and adding/overriding operations
 - *Polymorphism* = ability of a variable to reference objects from different classes at different times
 - *Dynamic binding (“dispatching”)* = interpretation of operation applied to polymorphic variable based on current class of referenced object
- } *Object-Oriented Design (“OOD”)*
also known as
Object-Based Programming

10

AdaCore OO Technology & Security

Why consider OOT for high-security software?

- Data-centric approach eases maintenance of many large systems
 - Inheritance is an effective technique for component reuse
- Model-driven architecture / UML tools may generate OO code to be certified
- Many programmers know OO languages such as C++, Java, or Ada 95
- Languages (such as Ada) used for high-integrity systems have OO features
- May want to take OO legacy code and apply CC/CEM *à posteriori*

What’s the catch?

- Paradigm clash
 - OOT’s distribution of functionality across classes, versus CC’s focus on tracing between requirements and implemented functions
- Technical issues
 - The features that are the essence of OOT complicate security certification
- Cultural issues
 - TOE evaluation personnel are not necessarily language experts and may (rightfully) be concerned about how to deal with unfamiliar technology

11

AdaCore Languages and CC/CEM – Basic Issues

No general-purpose language is appropriate at high EALs (6 and 7)

- Conflicts with reliability, predictability, analyzability

Key issue: can the language be subsetted to ease certification effort for TSFs/TOEs restricted to the subset

- Is/are the subset(s) precisely defined?
- Can compliance with the subset(s) be enforced by an automated tool?
- Who defines the subset(s)?
 - Standards agency? Tool/compiler vendor? Application developer?
- Are there intrinsic problems with the language that subsetting cannot solve?

Remainder of this presentation looks at current approaches

- C / C++
 - MISRA C
- Ada
 - SPARK
- Java
 - Real-Time / Safety-Critical Java Technology (JSRs 001, 302)

12

AdaCore MISRA C – Introduction

What is MISRA C?

- Reference document is *MISRA-C:2004 Guidelines for the use of the C language in critical systems* (October 2004)
- Subset of 1990 ISO C standard intended for embedded automotive systems up to and including Safety Integrity Level 3 (“SIL 3”)
- Comprises 141 rules (121 required, 20 advisory), supersedes 1998 version

Goals

- Promote safest possible use of C (not to promote C)
- Prevent or restrict usage of C constructs with unpredictable behavior
- Encourage production of static checking tools that enforce compliance with the subset

Some sample rules

- 9.1 (required) Assign a value to each automatic variable before using it
- 12.13 (advisory) Don't mix ++ or -- with other operators in an expression

“MISRA” = Motor Industry Software Reliability Association

13

AdaCore Challenges in Using C for High-Integrity Systems (*)

“The programmer makes mistakes”

- Small syntactic/lexical effects (e.g. from text entry errors) may have large semantic impact
- Weak type checking means many kinds of errors are undetected

“The programmer misunderstands the language”

- Complex rules for operator precedence, type conversions

“The compiler doesn’t do what the programmer expects”

- Annex G lists 201 features that are not completely defined or have behavior that can vary across implementations: *unspecified, undefined, implementation-defined, locale-specific*

“The compiler contains errors”

- Compiler writers may misinterpret the standard

“Run-time errors”

- C does not provide run-time checking for array indexing, validity of addresses for pointers, divide by zero, arithmetic overflow

(*) From MISRA C, §1.2, pp. 1-2

14

AdaCore MISRA C Assessment

Strengths

- Codifies “best practices” for C programming
- C smaller than other languages
 - Avoids OOP problems
- Large population of candidate users
- Wide assortment of tools, service providers

Weaknesses

- The base language, C, was not designed to meet the needs of high-integrity systems: errors of commission and omission
 - Attempting to support high reliability by subsetting goes against the grain of the language
- C (and thus MISRA C) does not readily “scale up” to large systems
- Some rules are not enforceable by static tools
- MISRA C is not a standard, and different tools enforce the subset differently
 - “The onus is on the user of this document to demonstrate that ... [the chosen vendor’s] tool set enforces the rules adequately” (§3.2, p. 6)

15

Why consider C++ for high-security systems?

- Interest in using legacy C++ code for new systems
- Programmer familiarity with the language
- “Safe” C++ subsets such as Lockheed-Martin’s coding standards for JSF

MISRA C++

- C++ Working Group under MISRA has been looking at issues
- Work in progress to develop a set of rules for C++ similar to what was done for C
 - Eliminate unpredictable behavior
 - Avoid common errors
 - Draft planned for mid-2007

When completed, this will likely have many of the same issues as MISRA C, and OOP will present challenges

- C++ subset guideline documents do not help solve the essential certification problems associated with OOP
 - They reduce some of the complexity but do not address the underlying issues

What is Ada?

- General-purpose methodology-neutral strongly-typed ISO standard language
- Intended for large, long-lived, reliability-critical embedded real-time applications

Ada in a nutshell

- Pascal-style foundation (syntax, basic data types)
- Exception handling
- Packages / encapsulation
- Generic templates
- Concurrency support (“tasking”)
- Object-Oriented Programming
 - Java-like interface feature added in Ada 2005
- Specialized support for particular domains, including:
 - Systems programming
 - Real-time systems
 - High-Integrity applications



AdaCore Ada for High-Security Applications

Full Ada not appropriate

- Run-time library too large / complex
- High level features interfere with traceability and analyzability
- Tradeoffs with other objectives (e.g, efficiency) led to some features with implementation-dependent or unspecified semantics

Why consider Ada for high-security software

- General design philosophy promotes sound software engineering
- Successful history with Ada 83 and Ada 95 for safety-critical systems
- Standard support for high-security systems
 - High-Integrity Systems Annex
 - User-defined tailoring, specifying features that are not used
 - Ravenscar profile
 - Restricted set of tasking features that are amenable to certification
- Guidelines documents
 - *Guide for Ada in High-Integrity Systems* (an ISO Technical Report)
 - *Guide for Ada Ravenscar Profile in High-Integrity Systems*

18

AdaCore Ada Assessment

Strengths

- Sound base language for programming high-reliability systems
- Allows construction of tailored subsets
- Scales up to large systems
- ISO Standard
 - Most recently, Ada 2005 standard published in March 2007
- Concurrency support / Ravenscar Profile
- Free and publicly available documents
- Good track record in safety-critical domains

Weaknesses

- Whole language is too large, so must be subsetted
 - Aside from the standard Ravenscar Profile, the exact subsets supported are vendor specific
- Language has some features with implementation-dependent or unspecified behavior
- Smaller user / tool vendor community than other languages

19

AdaCore **SPARK – Introduction**

SPARK = language + static analysis tools based on underlying principle of *correctness by construction*

- Facilitate rigorous, static demonstration that program does what (and only what) it is supposed to do
- Guarantee bounded time and space requirements
- Allow simple run-time library amenable to certification

Ada-based language

- Ada 95 + annotations – features that interfere with above principle
- Annotations are special comments handled by the SPARK tools

Language restrictions (Ada features intentionally omitted)

- Features that complicate analysis / formal proofs, e.g.:
 - Exceptions, goto statement, generic templates, function side effects
- Features that interfere with bounded time/space requirement, e.g.:
 - Recursion, pointers, dynamically-sized arrays

20

AdaCore **SPARK Summary**

Language features include:

- Most of Ada 95's "static semantic" features, including packages, private types, unconstrained array types, "OOP Lite"
- Concurrency (Ravenscar tasking profile)

Annotations

- Data / information flow (use of global variables, formal parameters)
- Inter-module dependencies
- Specification of dynamic behavior (pre/post-conditions, assertions)

Analysis performed by SPARK tools

- Static semantic analysis – detect aliasing, function side effects, ...
 - Data-flow analysis – detect unused or uninitialized variables, ...
 - Information-flow analysis – check input / output variable coupling
 - Verification condition generation – generate theorems
 - Theorem proving
- } *required*
- } *optional*

21

Strengths

- Language rules are unambiguous, implementation independent
 - SPARK code may be compiled by any standard Ada compiler
- Insecurities (e.g., “aliasing” of formal parameter and global variable) and many kinds of hard-to-detect bugs (e.g. use before initialization) are prevented
- Supports concurrency (Ravenscar profile)
- Positive experience across a range of high-integrity projects
 - Lower certification costs and post-delivery bug rates
- Directly matches requirements for formal techniques at EALs 6 and 7
- Excellent reference material

Weaknesses

- Language lacks some useful features
- Annotations require development style that may be unfamiliar
 - Attempting to take existing code and “SPARK”ing it is difficult
- Relatively small user community
- Small number of suppliers for tools, services

22

What is Java?

- A dynamic object-oriented programming language that includes exception handling and concurrency
 - Recent enhancements include templates, annotations
- A virtual machine (JVM) that serves as an execution engine
- An extensive class library (API)

Why consider Java for high-critical applications

- Generally well-defined semantics for sequential features
- It worries about some relevant issues
 - Attention to security, bytecode verification, index range checks
 - No references to uninitialized variables
 - No “dangling references”
- Real-Time Specification for Java (RTSJ) adds deterministic semantics and predictability for the threading features, and for memory areas not subject to Garbage Collection
- Work in progress on Safety-Critical Java Technology (JSR-302)

23

Strengths

- Incorporates specific security policies and mechanisms

Weaknesses

- Incorporates specific security policies and mechanisms

“Pure” OO language

- No “free” operation ⇔ system-managed memory reclamation / “Garbage Collection” (GC)
- Garbage collection interferes with predictability and/or adds latency
- The analyzability issues from OOT arise with Java

Complexity

- Language semantics (exceptions, threading, memory management, ...)
- Class library

Non-traditional run-time environment and execution model

- Dynamic loading, JIT compiling conflict with traditional compile/link/execute model
- JVM as execution engine (an interpreter) confuses the distinction between program and data
 - How to certify a system that includes a JVM

AdaCore Java and Safety-Critical Requirements (1)

Reliability

- Addresses many of the insecurities of C and C++
 - Run-time checks for array index out of bounds, etc.
 - Automatic garbage collection (but this interferes with predictability, analyzability)
 - No dangling references
- Annotation in Java 1.5 prevents accidental inheritance
- But there are a number of problems
 - Weak typing of primitives
 - C-based lexical structure and syntax
 - Example: `x==y` as a statement or `x=y` as an expression
 - Example: `0XF000000000000000` versus `16#F000_0000_0000_0000#`
 - Signed integer arithmetic will wrap around rather than overflow
 - Inheritance issues (accidental overriding)
 - Low-level (error-prone) thread model
 - Absence of named parameter associations

26

AdaCore Java and Safety-Critical Requirements (2)

Predictability

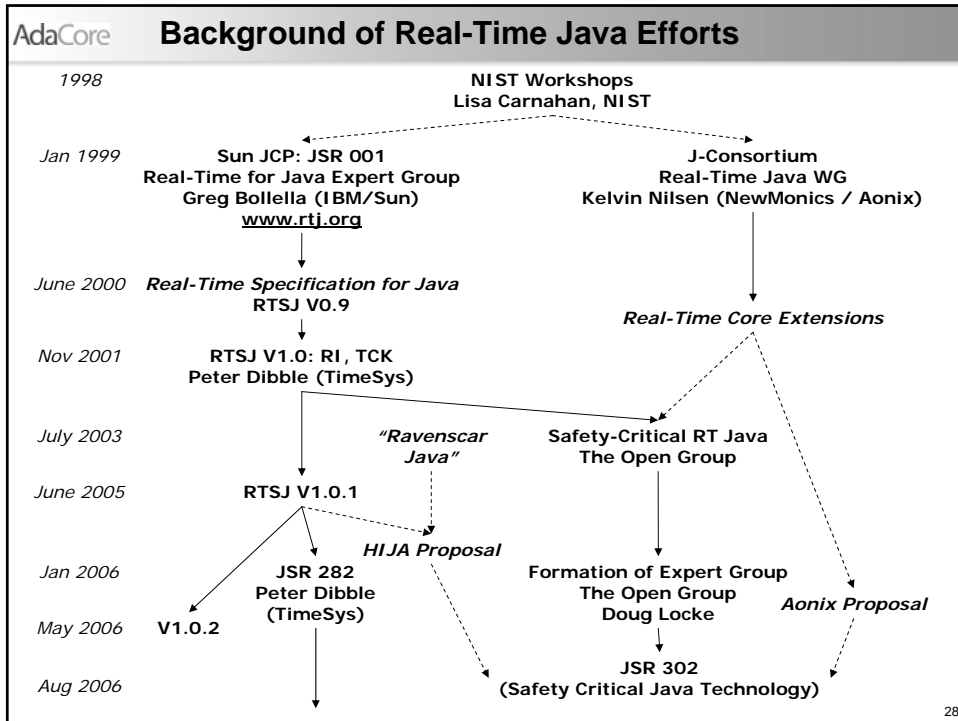
- Sequential Java is well-defined, except for semantics of finalization
- Thread model is underspecified
- Language definition is not a formal standard
- Garbage collection issues

Analyzability

- “Pure” OO language
 - Almost all of the analyzability issues identified above apply, except
 - Interface feature avoids the problems with general multiple inheritance
- Class combines “interface” and “implementation”, complicating analyzability
- Garbage collection issues
- Too complex for safety certification
 - Language semantics (exceptions, threading, memory management, ...)
 - Class library
 - Java Virtual Machine

Expressiveness

27



AdaCore Real-Time Specification for Java (1)

Goals

- Add real-time predictability to Java platform
 - Well-defined scheduling semantics
 - Priority inversion management
 - Avoid Garbage Collection latency
- Provide flexibility
 - Alternative schedulers
 - Dynamic effects (e.g. priority changes)

Concurrency

- Class `RealtimeThread` extends `java.lang.Thread`
 - Release parameters, scheduling parameters
- NoHeap real-time thread can preempt GC
- Flexible scheduling framework with on-line feasibility analysis
- User can supply handlers for deadline miss, cost overrun
- Base scheduler (fixed-priority, ≥ 28 priorities, FIFO within priority, preemptive)
- Support for periodic, aperiodic, sporadic real-time threads

AdaCore Real-Time Specification for Java (2)

Memory management

- Garbage-collected heap
- Immortal memory
- Scoped memory areas
 - A scoped area is used for allocations performed during a specified method
 - Reclaimable when no threads reference it
 - Run-time check needed when assigning a reference to a field of an object
 - Scoped memory is complicated and there are implementations that provide real-time garbage collection instead or in addition

Asynchrony

- Asynchronous Transfer of Control
- Asynchronous Events, Async Event Handlers

Time and timers

- High-resolution time (absolute, relative)
- Timers (periodic, one-shot)

Low-level features

30

AdaCore RTSJ and Safety-Critical Requirements

The RTSJ was never intended for safety-critical applications

- It is defined assuming the full generality of the Java language
- Some features (e.g., Asynchronous Transfer of Control) are too complex
- Many rules require run-time checks

But it does address some of the problems with full Java

- Garbage collection latency
- Underspecified semantics for thread scheduling
- Priority inversion management

There is work in progress to define a safety-critical real-time Java profile “based on” the RTSJ

- Started in July 2003 - The Open Group’s Real-Time Embedded Systems Forum
- Two main sources for ideas
 - HIJA (High-Integrity Java Applications) proposal from aicas (James Hunt)
 - Aonix *Scalable Real-Time Java* proposal (Kelvin Nilsen)
- Current status
 - Doug Locke is the spec lead (as of January 2006)

31

AdaCore **Safety-Critical Java Technology (JSR-302) Summary (1)**

Application Structure

```

graph LR
    Start([Start]) --> Init[Initialization]
    Init --> Mission[Mission Phase]
    Mission --> Rec[Recovery]
    Rec --> Halt([Halt])
    Rec --> Init
  
```

- Start phase creates “Mission Memory” scope
- Initialization phase allocates objects in Mission Memory
- Mission phase enters the mission scope and references, does not allocate, objects in Mission Memory
- Mission Memory reset (all objects reclaimed) at Recovery phase
- Application startup will not require heap memory

Garbage Collector

- None required

Scoped memory

- Nesting restricted
- Reference safety to be statically analyzable

Asynchrony

32

AdaCore **Safety-Critical Java Technology (JSR-302) Summary (2)**

All Schedulable Objects will be non-heap

Three Compliance Points (Levels 0, 1, 2)

- Level 0 provides a cyclic executive (single thread), no wait/notify
- Level 1 provides a single mission with multiple schedulable objects, no wait/notify
- Level 2 provides nested missions with (limited) nested scopes

Other features

- No Finalizers
- Requires Priority Ceiling for priority inversion management
- Priority Inheritance not required
- Class loader not required

Some issues being discussed

- Periodic RealTimeThreads vs. AsyncEventHandlers
 - Keep periodic threads, or
 - Just use asynchronous event handlers
- Use of annotations and offline pre-runtime analysis
- Scope nesting restrictions

33

Reliability

- Profiles' restrictions address a few of the issues raised by full Java, but most are intrinsic to the use of Java

Predictability

- Profiles address Java's issues with thread model, Garbage Collection
- Aonix proposal handles storage determination issue
- Java's (lack of) official standardization status applies to the profiles

Analyzability

- Annotations assist static analysis
- Profiles address some of Java's analyzability issues
 - Thread model, garbage collection
- Profiles also address some analyzability issues raised by the RTSJ
 - Eliminate ATC, dynamic priority changes, etc.
 - Require Priority Ceiling Emulation
- But the OOP analyzability issues are intrinsic to Java

Expressibility

34

The choice of programming language matters, and the various candidate languages have different strengths and weaknesses

MISRA C

- ⊙ Addresses the most serious deficiencies in C
- ⊙ Appropriate for smaller-sized systems
- ⊙ Large user/vendor base
 - but
- ⊙ Still based on an intrinsically insecure language, and does not readily "scale up" for large systems

Ada

- ⊙ Allows user-specifiable subsets
- ⊙ Firm foundation built on software engineering principles
- ⊙ Best fit technically to requirements for safety-critical OOP
- ⊙ Allows concurrency
 - but
- ⊙ Small user/vendor base

35

AdaCore **Summary (2)**

SPARK

- ⊗ Expressly designed to support rigorous methods, static analysis
- ⊗ Proven track record in high-integrity systems
- ⊗ Includes concurrency support
 - but
- ⊗ Small user/vendor base, rather restricted feature set

C++

- ⊗ Large potential user community
- ⊗ Better than C for large systems
 - but
- ⊗ Work on MISRA C++ is just beginning
- ⊗ Intrinsic issues due to C underpinnings, OOP

Safety-Critical Java Technology (JSR-302)

- ⊗ Serious interest from certain segments of safety-critical community
- ⊗ Secure language foundation
 - but

36

AdaCore **Summary (3) – “Report Card” on Languages**

Subjective ratings of language technologies for safety-critical systems

	MISRA C	C++	Ada	SPARK	Java
Reliability	Fair	Fair	Very Good	Excellent	Good
Predictability	Good	Good	Very Good	Excellent	Good
Analyzability	Fair	Fair	Good	Excellent	Good
Expressiveness	Fair	Good	Very Good	Fair	Fair
Size of user/vendor base	Very Good / Good	Very Good / Fair	Fair / Fair	Fair / Fair	Very Good / Fair

The ratings for C++, Ada and Java apply to tailored subsets rather than to the full languages

37

AdaCore Conclusions on OOT in Safety-Critical Systems

There is an essential conflict

- The features that make OOT attractive in general for software development complicate safety certification
- As system size increases, need language features and development methods that can help manage complexity, and OOT is extremely useful

Programming style can mitigate some of the OOT certification issues

- Use a subset of OO features

Language features can mitigate some of the OOT certification issues

- Encapsulation need not prevent coverage analysis of modules with hidden state
- Syntax can indicate intent of overriding or not overriding

Qualified tools can mitigate some of the OOT certification issues

- Transform dynamic binding into switch/case statements

“Bottom line”

- OOT inevitable in some types of safety-certified systems, including Level A
- Candidate OO languages offer tradeoffs in terms of technical advantages, language popularity, research interest

38

AdaCore Future Directions

Certification standards evolving to account for new language technologies

- Recognition of OOT, other modern features

Languages evolving to address safety certification requirements

- Profiles (subsets), support for rigorous methods

Tradeoffs will always exist in choosing a language

- Technical merit – expressibility, consistency with safety-critical requirements
- Cost of certification
- Availability of tools, programmers
- DER's experience with the language on previous systems

The distance between “state of the art” and “state of the practice” is shrinking

39

AdaCore Web Resources and Other References (1)

DO-178B

- RTCA SC-167 / EUROCAE WG-12. RTCA/DO-178B – *Software Considerations in Airborne Systems and Equipment Certification*, December 1992.

CAST Position Papers

- www.faa.gov/aircraft/air_cert/design_approvals/air_software/cast/cast_papers/

WG-71/SC-205 Web Site

- ultra.pr.erau.edu/SCAS/

Presentation by G. Ladier, “From DO-178B/ED-12B to DO-167C/ED-12C in civil avionics: why, how?”

- <ftp://estec.esa.nl/pub/wm/wme/bssc/Workshop/3-1.pdf>

OOT and Certification

- *Handbook for Object-Oriented Technology in Aviation (OOTiA)*, October 2004. www.faa.gov/aircraft/air_cert/design_approvals/air_software/oot
- F. Gasperoni. “Safety, Security and Object-Oriented Programming”. *Workshop on Innovative Techniques for Certification of Embedded Systems*, San Jose, California, April 4, 2006. (*)

(*) Available at www.adacore.com/category/developers-center/reference-library/technical-papers 40

AdaCore Web Resources and Other References (2)

MISRA C

- www.misra-c.com

C++

- Lockheed Martin, *Joint Strike Fighter Air Vehicle C++ Coding Standards for the System Development and Demonstration Program*, Doc. Number 2RDU00001 Rev C, Dec. 2005.
- S. Dewhurst, *C++ Gotchas: Avoiding Common Problems in Coding and Design*. Addison-Wesley, 2003.

MISRA C++

- www.era.co.uk/assc/wgssystem.asp

Ada

- <http://www.adaic.org>
- <http://std.dkuug.dk/JTC1/SC22/WG9/n359.pdf>
 - Guide for the Use of Ada in High-Integrity Systems
- A. Burns, B. Dobbing, T. Vardanega, *Guide for the use of the Ada Ravenscar profile in high integrity systems*; Univ. of York Technical Report YCS-2003-348;

(*) Available at www.adacore.com/category/developers-center/reference-library/technical-papers 41

SPARK

- <http://www.sparkada.com>
- J. Barnes, *High Integrity Software – The SPARK Approach to Safety and Security*, Addison-Wesley, 2003

Real-Time Specification for Java

- JSR-1: jcp.org/en/jsr/detail?id=1
- JSR-282: jcp.org/en/jsr/detail?id=1
- P. Dibble (spec. lead), R. Belliardi, B. Brosgol, D. Holmes, and A. Wellings. *Real-Time Specification for Java™, V1.0.1*, June 2005. www.rtsj.org

Safety-Critical Real-Time Java

- JSR-302 (Safety-Critical Java Technology): jcp.org/en/jsr/detail?id=302
- J. Kwon, A. Wellings, and S. King. *Ravenscar-Java: a high-integrity profile for real-time Java. Concurrency and Computation: Practice and Experience*, 17(5-6):681–713, April/May 2005.

University of York (UK) Safety-Critical Mailing List Archives

- www.cs.york.ac.uk/hise/safety-critical-archive

42

Security

- D. Wheeler, *Secure Programming for Linux and Unix HOWTO*, www.dwheeler.com/secure-programs/
- D. Herrmann
- R. Seacord
- Security Code Guidelines, java.sun.com/security/seccodeguide.html

43

AdaCore	Acronyms
CAST	Certification Authority Software Team
DER	Designated Engineering Representative (FAA-authorized auditor of compliance with DO-178B)
EUROCAE	European Organisation for Civil Aviation Equipment
GC	Garbage Collection
JCP	Java Community Process
JSR	Java Specification Request
MC/DC	Modified Condition/Decision Coverage
MISRA	Motor Industry Software Reliability Association
OOP	Object-Oriented Programming
OOT	Object-Oriented Technology
OOTiA	Object-Oriented Technology in Aviation
SIL	Safety Integrity Level
RTCA	<i>[Not an acronym, this is the name of an organization]</i>
RTSJ	Real-Time Specification for Java

44

AdaCore	Acronyms and Abreviations
IEC	International Electrotechnical Commission
ISO	International Organization for Standardization
IT	Information Technology
DER	Designated Engineering Representative (FAA-authorized auditor of compliance with DO-178B)
EUROCAE	European Organisation for Civil Aviation Equipment
GC	Garbage Collection
JCP	Java Community Process
JSR	Java Specification Request
MC/DC	Modified Condition/Decision Coverage
MISRA	Motor Industry Software Reliability Association
OOP	Object-Oriented Programming
OOT	Object-Oriented Technology
OOTiA	Object-Oriented Technology in Aviation
SIL	Safety Integrity Level

45

The complete presentation may be found at
www.adacore.com/wp-content/files/attachments/BrosGolGiccaPresentation-SSTC2007.pdf