AdaCore
The GNAT Pro Company

AdaCore
The GNAT Pro Company

North American Headquarters:

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

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

European Headquarters:

46 rue d'Amsterdam
75009 Paris
France

+33-1-4970-6716 (voice)
+33-1-4970-0552 (FAX)

www.adacore.com

Safety and Security: An Analysis of Certification Issues and Technologies for High-Integrity Software:

SSTC 2009
Salt Lake City, Utah

Half-Day Tutorial
Monday morning, 20 April 2009

Ben Brosgol • brosgol@adacore.com

AdaCore
The GNAT Pro Company

AdaCore Outline

Part 1: High-Integrity (safety- and/or security-critical) software certification

- Safety standards
  - DO-178B summary
  - DO-178C status
- Security standards
  - Common Criteria and Common Evaluation Methodology
- Comparison of safety and security certification Issues

Part 2: High-Integrity requirements and programming language technologies

- Impact of safety and security certification on language choice
  - Categorization of language vulnerabilities
  - Requirements on language characteristics
  - Issues raised by Object-Oriented Technology
- Language technology assessment for High-Integrity systems
  - C and C++ based approaches (MISRA C, MISRA C++)
  - Ada-based approaches (Ada, SPARK)
  - Virtual Machine / managed code approaches
    - Java
    - C# and Common Language Infrastructure / .NET

1

AdaCore Safety-Critical Software: Background

What is "safety critical" software?

- Failure can cause loss of human life or have other catastrophic consequences

How does safety criticality affect software development?

- Regulatory agencies require compliance with certification requirements
- Safety-related standards may apply to the finished product, to the development process, or both

Choice of programming language has large impact on the cost of developing / certifying safety-critical systems

- Dilemma: modern features that help in developing maintainable systems interfere with safety certification
  - Examples: Object-Oriented Programming, generics, exceptions, concurrency
- Traditional approach: select subset (profile) amenable to safety certification
  - Chosen features depend on the analysis methods to be used (or dictate which methods are feasible)

2

Approaches to Safety Certification

Prescriptive

- Specify requirements on the *process* by which software is developed and fielded
  - Sound process adds confidence in soundness of result
- Most applicable in domains where accepted practice is well understood
  - Relatively little innovation
- Safety responsibility seems to be with regulator versus developer
- Example: DO-178B

Goal-based

- Developer provides *safety cases*
  - *Claims* concerning system's safety-relevant attributes
  - *Arguments* justifying those claims
  - *Evidence* backing up the arguments
- Applicable in domains where technology changes rapidly
- Safety responsibility is with developer (to demonstrate that system is safe)
- Example: UK Defense Standard 00-56
  - "A Safety Case is a structured argument, supported by a body of evidence, that provides a compelling, comprehensible and valid case that a system is safe for a given application in a given environment"

3

DO-178B Basics

Software Considerations in Airborne Systems and Equipment Certification*, December 1992, published by RTCA

Comprises a set of 66 "objectives" for production of software for airborne systems

- Reliability: System does what it is supposed to do ⇨ no failures
  - Can trace each requirement to its implementing code and verification
  - No missing functionality
- Safety: System does not do what it is not supposed to do ⇨ no hazards
  - Can trace each piece of code back to a requirement
  - No additional functionality, no "dead code"
- Requires appropriate configuration management, quality assurance

"Level" of software establishes which objectives apply

* RTCA (www.rtca.org) is a U.S. Federal Advisory Committee whose recommendations guide FAA policy

4

AdaCore
The GNAT Pro Company

Software Levels Based on System Safety Assessment

Level A

- Anomalous behavior ⇨ catastrophic failure condition
  - "prevent continued safe flight and landing"

Level B

- Anomalous behavior ⇨ hazardous / severe-major failure condition
  - "serious or potentially fatal injuries to a small number of … occupants"

Level C

- Anomalous behavior ⇨ major failure condition
  - "discomfort to occupants, possibly including injuries"

Level D

- Anomalous behavior ⇨ minor failure condition
  - "some inconvenience to occupants"

Level E

- Anomalous behavior ⇨ no effect on aircraft operational capability or pilot workload

5

Structure / Goals / Usage

DO-178B guidelines organized into three major categories, each with a specified set of output artifacts

- Software Planning Process
- Software Development Processes
- "Integral" Processes

Appears oriented around new development efforts

- But may be applied to previously developed software, COTS, etc.

Strong emphasis on traceability

Implies traditional / static program build model

- Compile, link, execute

Used by FAA to approve software for commercial aircraft

- Developer organization supplies certification material
- Designated Engineering Representative ("DER") evaluates for compliance with DO-178B

"In a nutshell, what does this DO-178B specification really do?"*

- "It specifies that every line of code be directly traceable to a requirement and a test routine, and that no extraneous code outside of this process be included in the build"*

---

* Esterel Technologies, DO-178B FAQs, www.esterel-technologies.com/do-178b/

6

Other Aspects of DO-178B

Not specific to aviation

- Could be applied, in principle, to other domains (medical devices, etc.)

Includes glossary of terms

- "dead code", "deactivated code", "verification", ...

Does not dictate

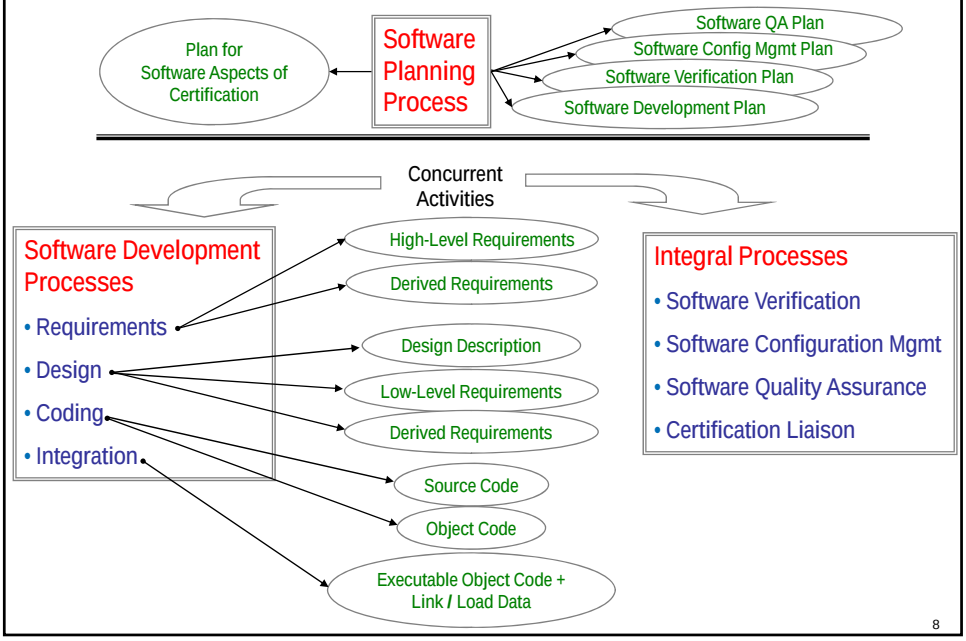
- Particular development process, design approach or notation
  - Modern software technologies (e.g. OOP) present some challenges
- Particular approach to hazard assessment (fault tree analysis, etc)
- Specific programming language(s) or software tools
- Requirements for personnel training
- Format for artifacts

What about security?

- No specific security-related objectives
- Security community has different culture and different standards
- In the future we may see some convergence, or the application of standards from one domain to systems from the other
- Details to be covered later

7

DO-178B Software Life Cycle Model



8

AdaCore
The GNAT Pro Company

AdaCore DO-178B and Programming Languages

DO-178B says very little about the choice of language

General principles (§4.4)

- "... choose requirements development and design methods, tools, and programming languages that limit the opportunity for introducing errors, and verification methods that ensure that errors introduced are detected"
- "... include safety features in the software design or Source Code to ensure that the software will respond correctly to input data errors and prevent output and control errors"

What's used in practice

- Assembly languages
- C (including MISRA subset), C++
- Ada 83, Ada 95, SPARK
  - Ada 2005 will probably start seeing usage
- Others (e.g. Fortran)
- Java is attracting interest in some areas
  - No Java system has yet been certified at any level higher than E, but this is likely to change

9

AdaCore Summary of DO-178B "Objectives"

	Safety Level			
Process	**A**	**B**	**C**	**D**
Software Planning Process	7	7	7	2
Software Development Process	7	7	7	7
Verification of Outputs of Software Requirements Process	3(ind) + 4	3(ind) + 4	6	3
Verification of Outputs of Software Design Process	6(ind) + 7	3(ind) + 10	9	1
Verification of Outputs of Software Coding & Integration Processes	3(ind) + 4	1(ind) + 6	6	0
Testing of Outputs of Integration Processes	2(ind) + 3	1(ind) + 4	5	3
Verification of Verification Process Results	8(ind)	3(ind) + 4	6	1
Software Configuration Management Process	6	6	6	6
Software Quality Assurance Process	3(ind)	3(ind)	2(ind)	2(ind)
Certification Liaison Process	3	3	3	3
Totals	66	65	57	28

Table shows number of objectives per process category
"ind" ⇨ need to show that objective is satisfied "with independence"

10

AdaCore Sample DO-178B Objectives [Table A-5]

Verification of Outputs of Software Coding and Integration Processes

Objective	*Level*	*Output*
Source Code complies with low-level requirements	ABC	Software Verification Results
Source Code complies with software architecture	ABC	
Source Code is verifiable	AB	
Source Code conforms to standards	ABC	
Source Code is traceable to low-level requirements	ABC	
Source Code is accurate and consistent	ABC	
Output of software integration process is complete and correct	ABC	

Underlining of level ⇨ "objective should be satisfied with independence"

11

AdaCore Some Issues related to Table A-5 Objectives [§6.3.4]

Reviews and Analyses of the Source Code

Conformance to standards

- Complexity restrictions
- Code constraints consistent with system safety objectives

Accuracy and consistency

- Stack usage
- Fixed-point arithmetic overflow and resolution
- Resource contention
- Worst-case execution timing
- Exception handling
- Use of uninitialized variables or constants
- Unused variables or constants
- Data corruption due to task or interrupt conflicts

12

AdaCore
The GNAT Pro Company

AdaCore Sample DO-178B Objectives [Table A-7]

Verification of Verification Process Results

Objective	*Level*	*Output*
Test procedures are correct	ABC	Software Verification Cases and Procedures
Test results are correct and discrepancies explained	ABC	Software Verification Results
Test coverage of high-level requirements is achieved	ABCD	
Test coverage of low-level requirements is achieved	ABC	
Test coverage of software structure (modified condition/decision coverage) is achieved	A	
Test coverage of software structure (decision coverage) is achieved	AB	
Test coverage of software structure (statement coverage) is achieved	ABC	
Test coverage of software structure (data coupling and control coupling) is achieved	ABC	

Underlining of level ⇨ "objective should be satisfied with independence"

13

AdaCore Role of Testing in Software Verification

Test cases are to be derived from software requirements

- Requirements-based hardware/software integration testing
- Requirements-based software integration testing
- Requirements-based low-level testing

Test cases must fully cover the code

- Unexercised code may be due to any of several reasons
  - Missing requirement ⇨ Add new requirement
  - Missing test ⇨ Add new test case
  - Dead code ⇨ Remove it
  - Deactivated code ⇨ Show that it will not be executed
- Coverage on source versus object code
  - May be demonstrated on Source Code for Levels B and below
  - May be demonstrated on Source Code for Level A unless compiler generates object code not directly traceable to Source Code
    - Then need additional verification on object code to establish correctness of such generated code

Structural coverage is not "white box" testing

- Need to show that all exercised code is traceable to requirements

14

AdaCore
The GNAT Pro Company

AdaCore Coverage Analysis Technology

Instrumentation-based

- Generated code produces coverage data
- Advantages
  - Direct correspondence with source code constructs
- Disadvantages
  - Instrumented code ≠ code that flies
  - Need to run the requirements-based tests twice (on the instrumented code and on the non-instrumented code) and show that they are equivalent

Non-instrumentation based

- Obtain coverage information from the running program, report in terms of source program
- Approaches
  - Debug script / execution monitor on target
  - Virtualization of target architecture on host
- Advantages
  - Analyzed code is the code that flies
- Disadvantages
  - Debug script approach can be slow

15

AdaCore Required Coverage Depends on Safety Level

	Level A	Level B	Level C	Level D
Data Coupling and Control Coupling *** Confirm uses of non-local data** *** Confirm invocations of subprograms**	✓	✓	✓	
Statement Coverage *** Every statement has been invoked at least once**	✓	✓	✓	
Decision Coverage *** Described below**	✓	✓		
Modified Condition / Decision Coverage *** Described below**	✓			

16

Data Coupling and Control Coupling

Definitions

- Data coupling: *The dependence of a software component on data not exclusively under control of that component*
- Control coupling: *The manner or degree by which one software component influences the execution of another software component*
- Component: *A self-contained part, combination of parts, sub-assemblies or units, which performs a distinct function of a system*

Intent of coverage analysis of data coupling and control coupling

- Assure correctness of components' interactions and dependencies
  - Components affect each other as, and only as, designer intended
- Needs to be conducted on requirements-based testing of integrated system
- Well-specified interfaces can make analysis easier

Example

- Requirement: in mode "No-Manual-Override", speed may be read but not updated by the Flight Control System
- May demonstrate compliance by showing that software cannot access routines that modify speed

17

Decision Coverage

"Condition"

- A Boolean expression containing no Boolean operators; e.g., `X>0`

"Decision"

- A Boolean expression composed of conditions and zero or more Boolean operators; e.g., `X>0 and Y=2`

Decision coverage

- Every point of entry and exit in the program has been invoked at least once
- Every decision in the program has taken all possible outcomes at least once

```
if X>0 and Y=2 then
   ... -- Branch 1
else
   ... -- Branch 2
end if;
```

Two tests sufficient for decision coverage
X = 1, Y = 2 ⇨ `True` ⇨ Branch 1
X = 0, Y = 2 ⇨ `False` ⇨ Branch 2

18

Modified Condition / Decision Coverage

MC / DC = Decision coverage + additional requirements

- Every condition in a decision in the program has taken all possible outcomes at least once
- Each condition in a decision has been shown to independently affect that decision's outcome

```
if X>0 and Y=2 then
   ... -- Branch 1
else
   ... -- Branch 2
end if;
```

Four tests needed for MC/DC
- X = 1, Y = 2 (True, True) ⇨ Branch 1
- X = 1, Y = 3 (True, False) ⇨ Branch 2
- X = 0, Y = 2 (False, True) ⇨ Branch 2
- X = 0, Y = 3 (False, False) ⇨ Branch 2

Fewer tests are required for decisions with short-circuit operators

```
if X>0 and then Y=2 then
   ... -- Branch 1
else
   ... -- Branch 2
end if;
```

Three tests needed for MC/DC
- X = 1, Y = 2 (True, True) ⇨ Branch 1
- X = 1, Y = 3 (True, False) ⇨ Branch 2
- X = 0, Y = * (False, *) ⇨ Branch 2

Benefit of MC/DC testing may be from the attention it forces on requirements specification, versus the actual test results

19

Costs of Certification at Different Levels

Experience of HighRely*

Level E Cost	Level D Cost	Level C Cost	Level B Cost	Level A Cost
Baseline	E + 5%	D + 30%	C + 15%	B + 5%

Why the increase from D to C?

- Testing low-level requirements
- Need for 100% coverage of source code statements
- Need for more rigorous review, configuration management

Added effort from C to B, or B to A, can be more modest

- Use qualified tools for coverage analysis

Main additional requirements for Level A

- Need for MC/DC testing
- Coverage at object code level if source not traceable to object
- Additional requirements for independence of developers (who need to meet objectives) from verifiers (who need to establish whether objectives are met)

* V. Hilderman & T. Baghai, *Avionics Certification: A Complete Guide to DO-178 (Software), DO-254 (Hardware)*, p. 40

20

Other Issues with DO-178B

Tool qualification

- Ensures that tool provides confidence at least equivalent to that of the process(es) eliminated, reduced or automated
- Can qualify as a *verification tool* (bug may fail to detect errors but won't introduce any) or as a *development tool* (bug may introduce errors)

Application of formal methods

- May augment but not replace the test/process-based approach otherwise required

Why is DO-178B process oriented?

- Quote from Gérard Ladier (Airbus) presentation at FISA-2003 conference
  - "It is not feasible to assess the number or kinds of software errors, if any, that may remain after the completion of system design, development, and test"
  - "Since dependability cannot be guaranteed from an assessment of the software product, it is necessary to have assurance on its development process"
  - *"You can't deliver clean water in a dirty pipe"*
- Another perspective (from presentation by John Rushby, HCSS Aviation Safety Workshop, Oct 2006)
  - "Because we cannot demonstrate how well we've done, we'll show how hard we've tried"

21

Certification Authority Software Team (CAST) (1)

International group of certification / regulatory authority representatives

- Seeks to promote harmonization of positions on software / hardware safety
- Publishes findings (clarification of DO-178B guidelines) in position papers www.faa.gov/aircraft/air_cert/design_approvals/air_software/cast/cast_papers/

CAST Number	Title of CAST Position Paper
CAST 1	Guidance for Assessing the Software Aspects of Product Service History of Airborne Systems and Equipment
CAST 2	Guidelines for Assessing Software Partitioning/Protection Schemes
CAST 3	Guidelines for Assuring the Software Aspects of Certification When Replacing Obsolete Electronic Parts Used in Airborne Systems and Equipment
CAST 4	Object-Oriented Technology (OOT) In Civil Aviation Projects: Certification Concerns
CAST 5	Guidelines for Proposing Alternate Means of Compliance to DO-178B
CAST 6	Rationale for Accepting Masking MC/DC in Certification Projects
CAST 7	Open Problem Report (OPR) Management for Certification
CAST 8	Use of the C++ Programming Language

22

AdaCore
The GNAT Pro Company

AdaCore Certification Authority Software Team (CAST) (2)

CAST Number	Title of CAST Position Paper
CAST 9	Considerations for Evaluating Safety Engineering Approaches to Software Assurance
CAST 10	What is a "Decision" in Application of Modified Condition/Decision Coverage (MC/DC) and Decision Coverage (DC)?
CAST 11A	Criteria for Assuring Continuous and Complete Software Verification Processes
CAST 12	Guidelines for Approving Source Code to Object Code Traceability
CAST 13	Automatic Code Generation Tools Development Assurance
CAST 14	[Use of a Level D Commercial Off-the-Shelf Operating System in Systems with Other Software of Levels C and/or D] *Withdrawn for Re-Evaluation*
CAST 15	Merging High-Level and Low-Level Requirements
CAST 16	Databus Evaluation Criteria
CAST 17	Structural Coverage of Object Code
CAST 18	Reverse Engineering in Certification Projects
CAST 19	Clarification of Structural Coverage Analyses of Data Coupling and Control Coupling
CAST 20	Addressing Cache in Airborne Systems and Equipment

23

AdaCore Certification Authority Software Team (CAST) (3)

CAST Number	Title of CAST Position Paper
CAST 21	Compiler-Supplied Libraries
CAST 22	Reuse of Software Tool Qualification Data Across Company Boundaries
CAST 23	Software Part Numbering
CAST 24	Reliance on Development Assurance Alone when Performing a Complex and Full-Time Critical Function
CAST 25	Considerations When Using a Qualifiable Development Environment (QDE) in Certification Projects
CAST 26	Verification Independence
CAST 27	Clarifications on the Use of RTCA Document DO-254 and EUROCAE Document ED-80, Design Assurance Guidance for Airborne Electronic Hardware
CAST 28	Frequently Asked Questions (FAQs) on the Use of RTCA Document DO-254 and EUROCAE Document ED-80, Design Assurance Guidance for Airborne Electronic Hardware
CAST 29	Use of COTS Graphical Processors (CGP) in Airborne Display Systems
CAST 30	Simple Electronic Hardware and RTCA Document DO-254 and EUROCAE Document ED-80, Design Assurance Guidance for Airborne Electronic Hardware

24

DO-178C

Organization / schedule

- Joint RTCA Special Committee 205 and EUROCAE* Working Group 71
  - "SC-205 WG-71", officially abbreviated as "SCWG"
  - Co-chairs: Jim Krodel (Pratt & Whitney), Gérard Ladier (Airbus)
- Semi-annual joint meetings in USA or Europe; started in March 2005
- Expected completion date: December 2008 (oops ☺)
- Web site (registration required)
  - ultra.pr.erau.edu/SCAS/

Subgroups

- SG1: SCWG Document Integration
- SG2: Issues and Rationale
- SG3: Tool Qualification
- SG4: Model Based Design and Verification
- SG5: Object-Oriented Technology
- SG6: Formal Methods
- SG7: CNS/ATM and Safety ("CNS" = Communications, Navigation and Surveillance)

* European Organisation for Civil Aviation Equipment (www.eurocae.org)

25

Why Revise DO-178B?

Address / accommodate "new" software technologies

- Object-Oriented Programming
- Model-based design / automatic code generators
- COTS software and tools, including real-time operating systems

Recognize that there is more to software verification than testing

- Formal methods
- Abstract interpretation

Take into account the supplementary papers / commentaries on DO-178B

- Certification Authorities Software Team (CAST) papers
- Issues Papers (IPs)

Try to remove/reduce the need for DERs to make subjective judgments

Correct errors

- Requirements on development tool qualification don't distinguish the host environment (in which the tool runs) from the target environment (in which the system runs)

26

AdaCore DO-178C "Terms of Reference"

Objectives

- Promote safe implementation of aeronautical software
- Provide clear and consistent ties with the systems and safety processes
- Address emerging software trends and technologies
- Implement an approach that can change with the technology

Activities (partial)

- Modify DO-178B
- Develop supplements to document technology-specific or method-specific guidance and guidelines
- Develop and document rationale for each DO-178B objective

Other considerations (partial)

- Maintain technology-independent nature of DO-178B objectives
- Modifications to DO-178B should:
  - Strive to minimize changes to existing text
  - Consider economic impact relative to system certification without compromising system safety
  - Address clear errors or inconsistencies / fill any clear gaps in DO-178B
  - Meet a documented need to a defined assurance benefit

27

AdaCore Controversy over DO-178C approach (1)

C. Michael Holloway (NASA), message posted to SC205/WG71 e-mail forum, 5 April 2005: *Are We Heading in the Right Direction?*

"The trend in the world (at least outside the U.S.) seems to be away from prescriptive standards and towards goal-based standards, which require the presentation of cogent arguments for safety (and other desired attributes), rather than the simple completion of certain prescribed processes.

This is a good trend. Although DO-178B/ED-12B has some attributes of a goal-based standard, for the most part it is a prescriptive standard of the sort that is increasingly difficult to justify on technical grounds.

The TOR's [Terms of Reference for DO-178C] insistence on minimizing changes to the current document is likely to ensure that DO-178C/ED-12C remains a mostly prescriptive standard. This is not, in my opinion, a good thing."

28

AdaCore
The GNAT Pro Company

AdaCore Controversy over DO-178C approach (2)

Martyn Thomas (Praxis), message posted to SC205/WG71 e-mail forum, 10 August 2005: *The Right Direction?*

"I strongly support the proposal that safety standards should move away from prescribing specific processes to requiring evidence that the delivered system has the required properties. The relationship between development processes and the properties of the resulting systems is simply too weak to justify any confidence that a system will be safe just because it has been developed in the specific way prescribed by some standard.

...the core of safety certification must be formal reasoning about properties (ideally assisted by automated analysis tools). That, in turn, requires that the safety requirements are stated with mathematical rigour.

...certification should depend on a mathematically sound argument that the delivered system has the necessary safety properties..."

29

AdaCore Rationale for DO-178C's Approach

Alternatives

- Major surgery on DO-178B
  - Remove prescriptive content, making document goal-oriented
  - Add supplements for requirements for specific approaches
- Minor surgery
  - Address new technologies in supplements when prescribed approaches may be replaced by others (e.g., formal verification)

Decision was to go with "minor surgery"

Reasoning

- DO-178B works
  - No commercial aviation deaths caused by software that had been certified to Level A
    - But there have been some close calls
- Experience with goal-based approach has been mixed
  - Change of mindset required for both developer and regulator
  - Safety cases tend to focus on individual hazards, but most accidents are due to a combination of factors

30

AdaCore OOT in DO-178C?

Should the revision to DO-178B contain explicit guidance related to the use of OOT?

- Yes
  - Without specific guidelines, certification will be too dependent on the subjective judgment of the DERs
  - OOTiA handbook is comprehensive reference and can serve as source of ideas
- No
  - DO-178B is intended to be independent of particular development methods and language features
  - Achieving consensus on a set of OOT-specific guidelines will be difficult

Prediction

- Major changes / enhancements are unlikely, but an annex may be supplied with guidelines on OOT issues

31

AdaCore Common Criteria / Common Evaluation Methodology

Security concern for an Information Technology (IT) product

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

International standards for 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

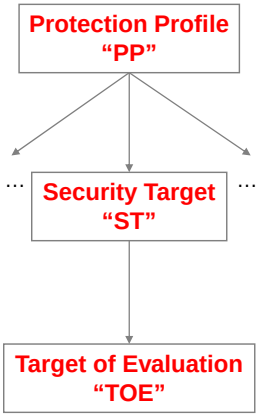
Status

- Current version is 3.1 (September 2007)

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

32

Common Criteria Artifacts



Consumer-specified implementation-independent specification of security requirements:
- Identify / characterize assets to be protected
- 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 or 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
- Evaluation conducted by an accredited Common Criteria Testing Lab, reviewed by National Evaluation Authority

33

Security Functional Requirements

Security Audit (FAU)
- Keep track of security-relevant activities and who initiated them

Communication (FCO)
- Assure identities of parties participating in data exchange (proof of origin/receipt)

Cryptographic Support (FCS)
- Manage keys, perform cryptographic functions

User Data Protection (FDP)
- Enforce confidentiality (access control) and integrity, within TOE and during communication

Identification and Authentication (FIA)
- Establish and verify a claimed user identity

Security Management (FMT)
- Manage security attributes, TSF data, and functions

Privacy (FPR)
- Protect user against discovery / misuse of identity

Protection of the TOE Security Functions (FPT)
- Protect TSF code and data

Resource Utilization (FRU)
- Support availability of required resources

TOE Access (FTA)
- Control establishment of user session

Trusted Path / Channels (FTP)
- Create / maintain trusted link between TSF and user or other trusted IT products

34

Security Assurance Requirements

Protection Profile Evaluation (APE)

Security Target Evaluation (ASE)

Development (ADV)

- ADV_ARC: Security Architecture
- ADV_FSP: Functional specification
- ADV_IMP: Implementation representation
- ADV_INT: Internals
- ADV_SPM: Security policy modeling
- ADV_TDS: TOE design

Guidance Documents (AGD)

- AGD_OPE: Operational user guidance
- AGD_PRE: Preparative procedures

Life-Cycle Support (ALC)

- ALC_CMC: Configuration Management capabilities
- ALC_CMS: Configuration Management scope
- ALC_DEL: Delivery
- ALC_DVS: Development security
- ALC_FLR: Flaw remediation
- ALC_LCD: Life-cycle definition
- ALC_TAT: Tools and techniques

Tests (ATE)

- ATE_COV: Coverage
- ATE_DPT: Depth
- ATE_FUN: Functional tests
- ATE_IND: Independent testing

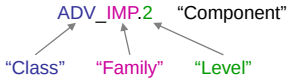
Vulnerability Assessment (AVA)

- AVA_VAN: Vulnerability analysis

Composition (ACO)

- ACO_COR: Composition rationale
- ACO_DEV: Development evidence
- ACO_REL: Reliance of dependent component
- ACO_CTT: Composed TOE testing
- ACO_VUL: Composition vulnerability analysis

Nomenclature for SFRs and SARs



35

Class ADV: Development

Purpose

- "Provide information about the TOE… used as the basis for conducting vulnerability analysis and testing upon the TOE"
- Show that "security functionality works correctly" and that "the TOE cannot be used … such that the security functionality can be corrupted or bypassed"

ADV_FSP: Functional Specification

- Definition of TSF interfaces

ADV_TDS: TOE Design

- Specification of TOE detailed design (modules, interfaces) showing TSF boundary

ADV_IMP: Implementation Representation

- Source code for TSFs

ADV_SPM: Security Policy Modeling

- Formal model of access control, audit policies, encryption policies, etc

ADV_ARC: Security Architecture

- Demonstration that security functions cannot be corrupted or bypassed

ADV_INT: TSF Internals

- Demonstration of well-structured / soundly engineered TSFs

36

Example of Security Assurance Requirement

ADV_IMP.2 **Complete mapping of the implementation representation**

Dependencies: ADV_TDS.3 Basic modular design
ALC_TAT.1 Well-defined development tools
ALC_CMC.5 Advanced support

Developer action elements:

ADV_IMP.2.1D The developer shall make available the implementation representation for the entire TSF.

ADV_IMP.2.2D The developer shall provide a mapping between the TOE design description and the **entire** implementation representation.

Content and presentation elements:

ADV_IMP.2.1C The implementation representation shall define the TSF to a level of detail such that the TSF can be generated without further design decisions.

ADV_IMP.2.2C The implementation representation shall be in the form used by the development personnel.

ADV_IMP.2.3C The mapping between the TOE design description and the **entire** implementation representation shall demonstrate their correspondence.

Evaluator action elements:

ADV_IMP.2.1E The evaluator ***shall confirm*** that the information provided meets all requirements for content and presentation of evidence.

37

Class ATE: Tests

Purpose

- "Provide assurance that the TSF behaves as described (in the functional specification, TOE design, and implementation representation)"

ATE_COV: Coverage

- Developer-supplied evidence of TSF coverage
  - "Coverage" means traceability (SFRs have been implemented), not code coverage

ATE_DPT: Depth

- Developer-supplied evidence of correct TSF behavior, based on ADV artifacts

ATE_FUN: Functional tests

- Developer-supplied test plans identifying the coverage and depth tests and their expected results
- Developer-supplied evidence from running the tests showing that the expected results have been obtained

ATE_IND: Independent testing

- Evaluator's analysis of developer test documentation, and performance of the tests

38

Class AVA: Vulnerability Assessment

Purpose

- Detect "exploitable vulnerabilities introduced in the development or the operation of the TOE"

AVA_VAN: Vulnerability analysis

- Different levels reflect rigor of analysis and assumed sophistication of attacker
- Developer only supplies the TOE for testing, the Evaluator conducts the assessment

AVA_VAN.1 Vulnerability survey (EAL 1)

- Evaluator conducts penetration testing assuming Basic attack potential

AVA_VAN.2 Vulnerability analysis (EAL 2, EAL 3)

- Evaluator performs independent vulnerability analysis based on TOE documentation and conducts penetration testing assuming Basic attack potential

AVA_VAN.3 Focused vulnerability analysis (EAL 4)

- Similar to ATE_VAN.2 but evaluator also uses TOE implementation representation, and assumes Enhanced-Basic attack potential

AVA_VAN.4 Methodological vulnerability analysis (EAL 5)

- Similar to ATE_VAN.3 but assumes Moderate attack potential

AVA_VAN.5 Advanced methodological vulnerability analysis (EAL 6, EAL 7)

- Similar to ATE_VAN.4 but assumes High attack potential

39

Evaluation Assurance Levels (EALs)

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

40

Higher EAL = More Secure?

SARs for a given EAL are relative to:

- Specific security functionality (SFRs)
- Operational environment

Example

- Various versions of Microsoft Windows (e.g. Windows 2000 at SP3) are certified at EAL4
- The applicable PP is the Controlled Access Protection Profile, Version 1.d ("CAPP")
  - niap-ccevs.org/cc-scheme/pp/pp.cfm/id/pp_os_ca_v1.d/
  - "The CAPP provides for a level of protection, which is appropriate for an **assumed non-hostile and well-managed user community** requiring protection against threats of **inadvertent or casual attempts to breach the system security**."
  - "The profile is **not intended to be applicable** to circumstances in which protection is required against **determined attempts by hostile and well-funded attackers** to breach system security."
  - "The CAPP does not fully address the threats posed by malicious system development or administrative personnel."

Bottom Line

- To understand the implications of a product's EAL, read the Protection Profile!

41

Composed Assurance Packages (CAPs)

CAP is EAL analog for a *composed TOE*, which consists of already-evaluated component TOEs

- Takes into account security issues arising from interactions among / interfaces to component TOEs

Each CAP implies a specific set of Security Assurance Requirements

Level	Description	Assurance
CAP-A	Structurally composed	Low - Moderate
CAP-B	Methodically composed	Moderate
CAP-C	Methodically composed, tested and reviewed	Moderate - High

"The CAPs only consider resistance against an attacker with an attack potential up to Enhanced-Basic. This is due to the level of design information that can be provided through the ACO_DEV [family], limiting some of the factors associated with attack potential (knowledge of the composed TOE) and subsequently affecting the rigour of vulnerability analysis that can be performed by the evaluator. Therefore, the level of assurance in the composed TOE is limited, although the assurance in the individual components within the composed TOE may be much higher."

[Common Criteria, Part 3, Version 3.1; p. 47]

42

DO-178B *vis à vis* Common Criteria

Some basic similarities

- Concern with full software development life cycle activities
- Tiered approach reflecting tradeoffs (system must be safe/secure enough)
- Strong emphasis on testing as a major element of software verification

Some important differences

- Scope of requirements
  - DO-178B: entire system
  - Common Criteria: only the security-related components
- Functional requirements
  - DO-178B: no specific set of "safety functions"
  - Common Criteria: Security Functional Requirements
- Application domain focus
  - DO-178B: real-time embedded
  - Common Criteria: "enterprise" IT products
- System users/operators
  - DO-178B: known personnel, trustworthy
  - Common Criteria: Unknown end-users, possibly threats

43

Comparison of Security and Safety Issues

Compliance with safety standards is in some sense more demanding

- Each component must be certified against the requirements for its 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

Compliance with security standards is in some sense more demanding

- Requirement for formal methods at highest EAL
- Vulnerability analysis is difficult and must assume sophisticated adversary

Increasing awareness of need for security in safety-critical systems

- FAA "Special Conditions" notice for B787-8 (Feb 2008)
  - Systems and Data Networks Security – Isolation or Protection from Unauthorized Passenger Domain Systems Access
- Study of issues related to dual certification
  - C. Taylor, J. Alves-Foss, B. Rinker; *Merging Safety and Assurance: The Process of Dual Certification for Software*; STC 2002

44

Part 2: Safety, Security, and Programming Languages

Impact of safety and security certification on language choice

- Categorization of language vulnerabilities
- Requirements on language characteristics
- Issues raised by Object-Oriented Technology

Language technology assessment for High-Integrity systems

- C-based approaches
  - MISRA C
  - C++ “safe subsets”
- Ada-based approaches
  - Ada
  - SPARK
- Virtual Machine / managed code approaches
  - Java
    - Real-Time Specification for Java (JSR-001, JSR-282)
    - Safety-Critical Java Technology (JSR-302)
  - C#

Conclusions

45

ISO/IEC JTC 1/SC 22/WG 23: Programming Language Vulnerabilities*

Purpose / Background

- Prepare Technical Report: “Guidance to Avoiding Vulnerabilities in Programming Languages through Language Selection and Use”
- Guidance spans multiple programming languages including C, C++, Java, Ada, Fortran
- Applicable to systems that are safety-critical, high-security, mission critical
- Influenced by MITRE work on Common Weakness/Vulnerability Enumeration (CWE, CVE)

Organization / Schedule

- Convenor: John Benito
- Secretariat: Jim Moore
- Representatives from ISO language standard WGs, non-ISO language communities, national standards bodies, governmental agencies
- Draft Technical Report (without language-specific annexes) planned for May 2009

Resources

- www.aitcnet.org/isai/
  - Home page with general project information, links to documents
- aitc.aitcnet.org/isai/_Mtg_10/22-WG23-N-0170/n0170.pdf
  - Preliminary Draft Technical Report (10 November 2008)

* Originally “ISO/IEC JTC 1/SC 22/OWGV (Other Working Group: Vulnerabilities)”

46

WG23 Preliminary Draft Technical Report (PDTR)

Terminology

- Language vulnerability
  - "A *property* (of a programming language) that can contribute to, or that is strongly correlated with, application vulnerabilities in programs written in that language"
- Application vulnerability
  - "A security vulnerability or safety hazard, or defect"
- Security vulnerability
  - "A weakness … that could be exploited or triggered by a threat"

Template for vulnerability enumeration

- Description
- Cross reference (to external documents or other vulnerabilities)
- Categorization
- Mechanism of failure
- Applicable language characteristics
- Avoiding the vulnerability of mitigating its effects
- Implications for standardization
- Bibliography

47

PDTR: Programming Language Vulnerabilities (1)

General

- Unspecified / undefined behavior
- Dead / deactivated code
- Leveraging human experience
- Deprecated language features
- Dynamically linked or self-modifying code
- Implementation-defined behavior
- Unspecified functionality

Lexical

- Pre-processor directives
- Choice of clear names

Variables

- Unused variable
- Initialization of variables

Numeric types

- Numeric conversion errors
- Wrap-around error
- Sign extension error
- Floating point arithmetic
- Bit representations

Arrays

- Buffer overflow in stack
- Buffer overflow in heap
- Boundary beginning violation
- Off-by-one error
- Unchecked array indexing

Pointers

- Null pointer dereference
- Dangling reference to heap
- Memory leak
- Pointer casting and pointer type changes
- Pointer arithmetic
- Dangling references to stack frames

Other data type issues

- Type breaking reinterpretation of data
- Type system
- Enumerator issues

48

AdaCore
The GNAT Pro Company

AdaCore PDTR: Programming Language Vulnerabilities (2)

Expressions
- Operator precedence / Order of evaluation
- Likely incorrect expressions
- Side effects and order of evaluation

Subprograms
- Subprogram signature mismatch
- Passing parameters and return values
- Recursion
- Returning error status

Statements
- Switch statements and static analysis
- Demarcation of control flow
- Loop control variables

Program Structure
- Identifier name reuse
- Templates and generics
- Mixed-language programming
- Termination strategy

Object-Oriented Programming
- Inheritance

API
- Use of libraries
- Choice of filenames and other external identifiers

Some omissions from the current WG23 list
- Concurrency (in progress)
- OOP vulnerabilities beyond inheritance

Focus on features means that some underlying issues are implicit
- General problem of failure to validate user input

Language-specific annexes?
- Under discussion

49

AdaCore Language Requirements Implied by DO-178B / CC (1)

Reliability
- Support for development of readable, correct code
  - No “traps and pitfalls”
  - Intuitive lexical and syntactic features
- Support for good software engineering practice
  - Encapsulation
  - Reliable concurrency features
- Early error detection
  - Strong typing
    - Avoid conversion errors
  - Compile-time checking whenever possible
  - Run-time checking for array indexing etc

Predictability
- Unambiguous language semantics
  - No undefined / implementation-dependent behavior

50

Language Requirements Implied by DO-178B / CC (2)

Analyzability

- DO-178B
  - General correctness properties
    - No references to uninitialized variables, unprotected accesses to shared data, etc.
  - Time/space predictability
  - Requirements coverage / bi-directional traceability
    - No dead code
- Common Criteria
  - Correctness of security functions' implementation
  - Absence of application vulnerabilities

Expressiveness

- General-purpose features consistent with the above requirements
- Support for specialized processing typical in embedded systems (safety)
  - Interrupt handling
  - Low-level programming
  - Fixed-point arithmetic
- Support for security-related functionality

51

Another View of Analyzability*

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*

52

AdaCore
The GNAT Pro Company

AdaCore Modern Language Features and DO-178B / CC

Many features help

- Encapsulation / namespace control ⇨ less data coupling
- Strong typing, compile-time checking ⇨ early error detection

Some features present difficulties

- Need for expressive power in general-purpose language, versus need to constrain feature usage in order to demonstrate compliance with safety/security standard
  - 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
  - Generated code may contain implicit loops, conditionals
  - Run-time library, API need to be certifiable
  - 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
  - Optimization may interfere with traceability

53

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")

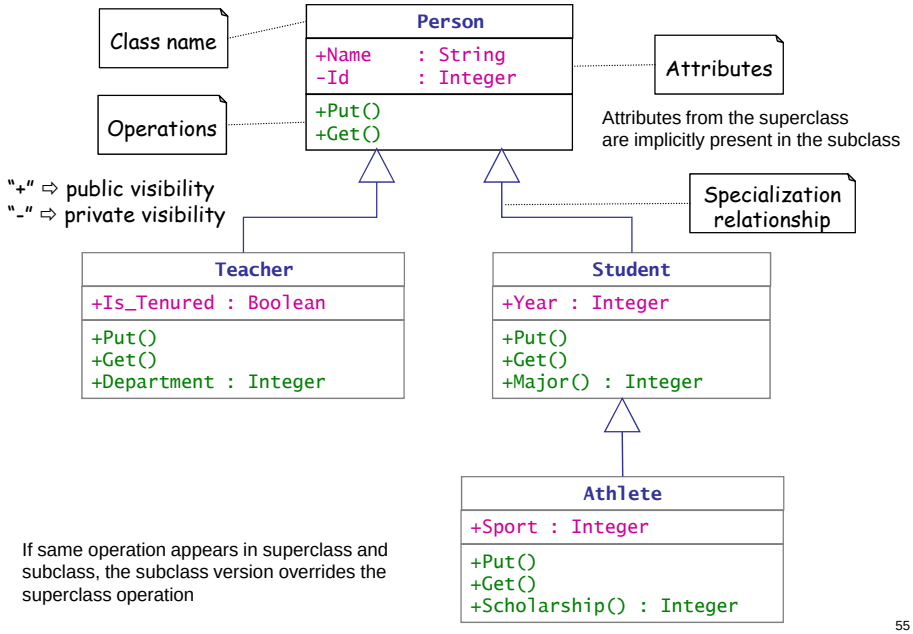
- *Object* = state ("attributes") + operations ("methods")
- *Class* = module + object creation template
- *Encapsulation* = separation of interface (spec for methods) from implementation (state, algorithms)

Object-Oriented Design ("OOD")
also known as
Object-Based Programming

- *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
  - Operation should behave correctly for an object from any class in the hierarchy ("Liskov Substitution Property")

54

Example of Inheritance Hierarchy in UML



55

Object-Oriented Technology ("OOT"), cont'd.

Additional OOP elements

- Single versus multiple inheritance
  - "Interface" for a simple form of multiple inheritance
- Use of constructors / destructors

Related language features

- Method overloading
- Type conversion
- Inline expansion

Other modern language features that complicate safety certification

- Generic templates
- Exceptions
- Concurrency

56

AdaCore
The GNAT Pro Company

AdaCore OO Technology & Safety/Security Certification

Why consider OOT for High-Integrity 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 safety-critical systems have OO features
- May want to take OO legacy code and apply DO-178B / Common Criteria *à posteriori*

What's the catch?

- Paradigm clash
  - OOT's distribution of functionality across classes, versus safety analysis's focus on tracing between requirements and implemented functions
- Technical issues
  - The features that are the essence of OOP complicate safety certification and raise security issues (e.g. ensuring integrity of "vtable")
- Cultural issues
  - Many DERs / TOE evaluation personnel are not language experts and are (rightfully) concerned about how to deal with unfamiliar technology

57

AdaCore "Object-Oriented Technology in Aviation" Project

Background

- DO-178B was written without OOT in mind
- Several current languages that might be used for safety-critical systems support OOP
  - Can the OO features be used?
  - If so, what are the certification-related issues and feature usage guidelines that developers and DERs need to be aware of?
- Several FAA/NASA-organized workshops have focused on these points, resulting in a 4-volume handbook

Status of OOTiA Handbook

- Most recent draft was published in Oct 2004
- Feedback will dictate eventual disposition
  - Possibilities include an FAA OOT Advisory Circular, or OOT guidance sections in DO-178C

58

AdaCore OOTiA Handbook

OOTiA handbook provides "input", not "guidance"

- Vol 1- Overview
- Vol 2- Considerations and Issues ("things to worry about")
- Vol 3- Best Practices
- Vol 4- Certification Practices ("things that DERs will ask")

From the *OOTiA Handbook*:

OOT in embedded and safety-critical systems is still an evolving discipline; best practices will likely be added and modified in the future revisions of this Handbook.

In all cases, it should be noted that it is still the developer's responsibility to demonstrate that the OOT methods and processes they have selected to utilize can, and will, provide the appropriate integrity for safe software implementation.

In any case, the OOT standards and methods the developer intends to use should be documented in the project plans and standards, and presented to the certification authorities as early as possible in the project.

59

AdaCore OOTiA Handbook – Summary of Issues

Single inheritance / dynamic dispatch

Multiple inheritance

Templates

Inlining

Type conversion

Overloading and method resolution

Dead and deactivated code, and reuse

OO Tools

Traceability

Structural coverage

60

AdaCore Issues with OOT and Safety / Security Certification (1)

Class

- Class-as-module ⇨ dead or deactivated code [Analyzability]

Encapsulation

- Simple "accessor" methods ⇨ inline expansion [Analyzability]
- Private data ⇨ coverage analysis [Analyzability]

Inheritance

- Unintended inheritance instead of overriding [Reliability]
  - Misspell the method name in a subclass
- Unintended overriding ("fragile base class") [Reliability]
  - Add method to superclass with same signature as in some subclass
- Incorrect interpretation of field name [Reliability]
  - Add field to intermediate subclass with same name as in superclass
- Reusability of superclass coverage tests for subclass [Analyzability]

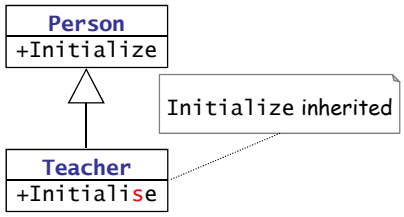
Polymorphism

- Pointers [Analyzability]
- Dynamic allocation [Analyzability]

61

AdaCore Example: Unintended Inheritance

Misspelling of method name in subclass

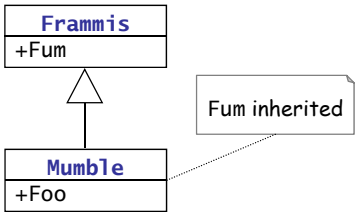


- Programmer meant to override `Initialize` but misspelled the name
- Superclass's `Initialize` is inherited rather than overridden
- Dynamic binding for `p.Initialize()`, when `p` (a polymorphic `Person`) references a `Teacher`, invokes the `Initialize` method defined for `Person`

62

Example: Unintended Overriding

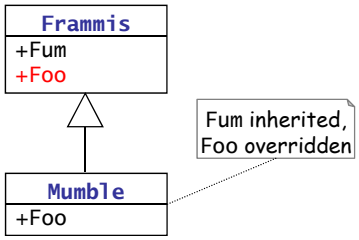
Initially:



- Method Foo is defined only for subclass Mumble

During maintenance:

- Foo is added to superclass Frammis

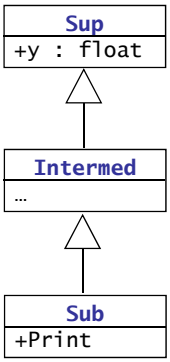


- Dynamic binding for `p.Foo()`, where p is a polymorphic Frammis and references a Mumble, invokes the Foo method previously defined for Mumble
- This is not the intended effect

63

Example: Incorrect Interpretation of Field Name

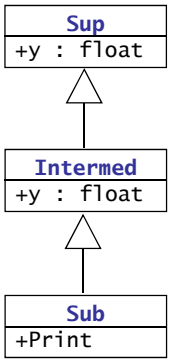
Initially:



- Method Print displays the value of field y, from class Sup

During maintenance:

- Field y is added to class Intermed



Some languages allow same name to be used for fields in different classes in same hierarchy

- Method Print now displays the value of y from class Intermed
- This is not the intended effect

64

AdaCore Issues with OOT and Safety / Security Certification (2)

Dynamic binding

- Determination of which method is invoked [Analyzability]
  - Harder than "switch/case" statement since alternatives are not localized to a given piece of program text
  - How to ensure test coverage: need to test each possible dispatch at each invocation point?
  - Complicates source/object traceability
- Verification of correctness of "dispatch vector" ("vtable") [Analyzability]
  - Proper initialization
  - No malicious/erroneous updates during program execution

Multiple inheritance

- Semantic complexity when inherit same method or field from different superclasses [Reliability, Analyzability]

Constructors / destructors

- Possibility of constructor referencing uninitialized data [Reliability]
- Semantics of destructors invocation [Predictability]

65

AdaCore Languages and DO-178B / CC – Basic Issues

No general-purpose language is appropriate for highest safety/security levels

- Conflicts with reliability, predictability, analyzability

Key issue: can the language be subsetted so that applications restricted to the subset can be certified

- 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 cannot be solved by subsetting?

Remainder of this presentation looks at current approaches

- C-based
  - MISRA C
  - C++
- Ada-based
  - SPARK
  - Ada (pragma Profile, pragma Restrictions)
- Java-based
  - Safety-Critical Java Technology (JSR-302)

66

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

Sources of the rules

- Annex G of ISO 9899 (C standard)
- *MISRA Development Guidelines for Vehicle Based Software*
- B.W. Kernighan and D.M. Ritchie, *The C Programming Language*, 2nd edition, 1988
- A. Koenig, *C Traps and Pitfalls*, 1988
- IEC 61508, *Functional safety of electrical/electronic/programmable electronic safety-related systems*,1998-2000

"MISRA" = Motor Industry Software Reliability Association

67

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
  - "=" versus "==", extra ";" on end of "if" statement
  - Leading '0' on octal literals
- Weak type checking means many kinds of errors are undetected
  - "If C is presented with a square peg and a round hole it doesn't complain, but makes them fit!"

"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

68

Outline and Sample Rules Paraphrased* (1)

1. Environment (5 rules)

- 1.2 (required) Don't depend on undefined or unspecified behavior

2. Language extensions (4 rules)

- Deals with assembly language ("encapsulate". "isolate") and comment conventions

3. Documentation (6 rules)

- 3.1 (required) Document all uses of implementation-defined behavior
- 3.3 (advisory) Determine and document how implementation deals with integer division

4. Character sets (2 rules)

- 4.2 (required) Don't use trigraphs (e.g. "??-")

5. Identifiers (7 rules)

- Rules disallow reuse of same identifier in different contexts, to prevent confusion

6. Types (5 rules)

- Dictates usage of char types, bit fields

7. Constants (1 rule)

- 7.1 (required) Don't use octal constants or octal escape sequences

* Need to paraphrase since MISRA FAQ says "you may not quote rule texts without our permission"

69

Outline and Sample Rules Paraphrased (2)

8. Declarations and definitions (12 rules)

- 8.5 (required) Don't define objects or functions in a header file

9. Initialisation (3 rules)

- 9.1 (required) Assign a value to each automatic variable before using it

10. Arithmetic type conversions (6 rules)

- Several rules dictate when implicit conversions are permitted, for integer and floating types
- Other rules cover casting, use of "U" suffix for constants of `unsigned` type

11. Pointer type conversions (5 rules)

- Rules prevent pointer conversions from introducing various type errors
- Don't cast away `const`

12. Expressions (13 rules)

- 12.1 (advisory) Make limited use of C's operator precedence rules
- 12.2 (required) Don't depend on order of evaluation
- 12.13 (advisory) Don't mix ++ or `--` with other operators in an expression

13. Control statement expressions (7 rules)

- 13.1 (required) Don't use assignment operators in Boolean expressions
- 13.3 (required) don't use == or != on floating-point values

70

Outline and Sample Rules Paraphrased (3)

14. Control flow (10 rules)

- Rules prohibit unreachable code, `goto` statements, `continue` statements
- 14.7 (required) A function may have only one point of exit
- 14.9 (required) Need to follow `if(`*expr*`)` by a compound statement, and an `else` by either a compound statement or another `if` statement

15. Switch statements (5 rules)

- Put a `break` at the end of each non-empty switch clause, and a `default` clause at the end

16. Functions (10 rules)

- No var-args, no recursion
- 16.10 (required) If a function returns an error code, the code needs to be checked

17. Pointers and arrays (6 rules)

- 17.1 (required) Only use pointer arithmetic on a pointer that references an array or array element
- 17.6 (required) Don't assign the address of an automatic object to a more global variable

71

Outline and Sample Rules Paraphrased (4)

18. Structures and unions (4 rules)

- 18.3 (required) Don't reuse the same area of memory for different purposes
- 18.4 (required) Don't use unions

19. Preprocessing directives (17 rules)

- Various rules on #include, #define, #undef
- 19.7 (advisory) Use functions instead of macros
- 19.15 (required) Don't include the same header file more than once

20. Standard libraries (12 rules)

- 20.2 (required) Don't reuse names of standard library entities
- 20.4 (required) Don't use dynamic heap memory
- Other rules indicate additional libraries and functions that are not to be used

21. Run-time failures (1 rule)

- 21.1 (required) Use static analysis, dynamic analysis, and/or explicit coding of checks, to minimize/handle run-time failures

MISRA-C document recognizes that there may be situations when some of the rules may need to be violated, and it gives suggestions on how to deal with such issues

72

AdaCore MISRA C Strengths

Codifies "best practices" for C programming

- General good style
- Avoids C features that are intrinsically non-portable / ill-defined

Simplicity of C compared with other languages

- Smaller run-time library ⇨ easier certification than run-time library for larger languages
- Avoids OOP problems

Can apply also to C++

- A C++ programmer can generally adhere to the MISRA C guidelines
- MISRA C is basis for MISRA C++

Large population of candidate users

- Natural migration path for embedded system developers already familiar with C

May be appropriate for small systems on specialized hardware lacking compilers for alternative languages

- MISRA-C compatible code may be produced as output of source-to-source translator

Wide assortment of tools, service providers

Safety-critical subset of MISRA C has been claimed

- C[b]

73

AdaCore MISRA C 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 lacks features such as encapsulation, namespaces (packages)
- Certification cost for MISRA C system is likely to be higher than for other languages designed with high reliability in mind

C (and thus MISRA C) does not readily "scale up" to large systems

C90 standard has ambiguities and thus so does MISRA C

Some rules are not enforceable by static tools

- What documentation is required for the rules in section 3?

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)
- Prohibiting dangling references (17.6) may be enforced through rules of varying degrees of conservatism

Usage of MISRA C (e.g. as basis for internal company standard) requires license from MISRA

74

AdaCore
The GNAT Pro Company

C++ and High-Integrity Software

Why consider C++ for safety- or security-critical systems?

- Interest in using legacy C++ code for such systems
- Programmer familiarity with the language
- Existence of "safe" C++ subsets

"JSF++"

- Lockheed Martin's C++ coding standards for Joint Strike Fighter Program
- 221 rules, serving several purposes
  - General good style
  - MISRA-C based rules
  - C++ specific restrictions, in particular on OOP features

MISRA C++

- Goals for C++ similar to what was done for MISRA-C
  - Eliminate unpredictable behavior
  - Avoid common errors
- Launched in June 2008

75

JSF++

JSF++ Strengths

- Codifies "best practices" for C++ programming
- Large population of candidate users
- Support by tool vendors

JSF++ Weaknesses

- C++ language issues
  - C foundation (see Ian Joyner's critique)
  - No language support for concurrency
- Issues with the rules
  - Some rules are not enforceable by static tools
  - Different tools can enforce some rules differently
  - JSF++ not widely reviewed
  - Does not solve the essential safety/security certification problems associated with OOP
    - Reduces some of the complexity but do not address the underlying issues
    - Almost all of the OOT issues identified earlier apply to C++, except
      - Use of "friends" can accommodate coverage testing of modules with hidden state

76

MISRA C++ (1)

What is MISRA C++?

- *MISRA C++:2008 Guidelines for the use of the C++ language in critical systems*
- Subset of 2003 version of C++
  - For "production code in critical systems"
- Contains 228 rules (199 required, 17 advisory, 12 documentary)
- Contains appendix categorizing C++ vulnerabilities
  - "Unspecified", "Undefined", "Implementation", "Indeterminate", "No diagnostic required"

Goals

- Promote safest possible use of C++ (not to promote C++)
- Enhance competence in C++ programming
- Encourage production of static checking tools that enforce compliance with the subset

Outline of document

1 Background
2 The vision
3 Scope
4 Using MISRA C++
5 Introduction to the rules
6 Rules
7 References
Appendix A: Summary of rules
Appendix B: C++ vulnerabilities
Appendix C: Glossary

77

MISRA C++ (2)

Sample rules

- Language
  - 1-0-3 (Doc) The implementation of integer division in the chosen compiler shall be determined and documented
- Identifiers
  - 2-10-6 (Req) If an identifier refers to a type, it shall not also refer to an object or a function in the same scope
- Linkage specifications
  - 7-5-4 (Adv) Functions should not call themselves, either directly or indirectly
- Declarators – initializers
  - 8-5-1 (Req) All variables shall have a defined value before they are used
- Multiple base classes
  - 10-1-2 (Req) A base class shall only be declared virtual if it is used in a diamond hierarchy
- Language support library – Dynamic memory management
  - 18-4-1 (Req) Dynamic heap memory allocation shall not be used

Critique

- Many of the issues with MISRA C apply also to MISRA C++
- Rules do not really address the difficult issues with OOP and safety certification

78

Ada – Introduction

What is Ada?

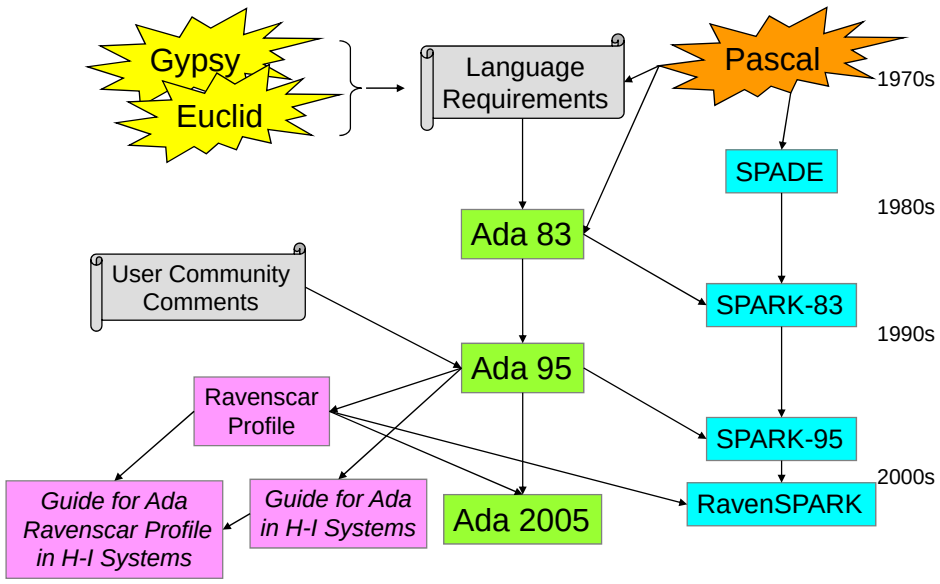
- General-purpose methodology-neutral strongly-typed ISO standard language
- Intended for large, long-lived, reliability-critical embedded real-time applications

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

79

Ada History from a High-Integrity View



80

AdaCore
The GNAT Pro Company

AdaCore Ada for High-Integrity 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 safety/security-critical software

- General design philosophy promotes sound software engineering
- Successful history with Ada 83 and Ada 95
- Standard support for safety/security-critical systems
  - Safety and Security Annex (renamed High-Integrity Systems Annex in Ada 2005)
  - User-defined tailoring, specifying features that are not used (pragma Restrictions)
    - Compiler rejects programs that use these features
    - Executable program excludes run-time libraries for these features
  - 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*

81

AdaCore Ada and High-Integrity Requirements (1)

Reliability

- Very few features have surprising effects
- Prevention of dangling references to declared objects, subprograms
- Specification of intent on operation inheritance
- Pragma Assert

} Ada 2005

- But: no garbage collection ⇨ programmer responsible for memory management

Predictability

- Language semantics are generally well-defined, in an ISO standard
  - But there are features whose effects are implementation defined, implementation dependent, unspecified, or “bounded errors”
- Implementation decisions can affect time or space predictability
  - Functions returning unconstrained arrays
- Solutions in practice
  - Analyze source program (e.g. no read of uninitialized object)
  - Adhere to subset (no functions returning unconstrained arrays)
  - Analyze object code so that implementation decision is known

82

Ada and High-Integrity Requirements (2)

Analyzability

- Some features help analyzability
  - Child units may be used for testing packages with encapsulated state
- But full language is too complex; need to subset
- Key features are `pragma Restrictions` and `pragma Profile`
  - Ada is in effect a family of profiles, where user can select features *à la carte*
  - No such thing as *the* safety-critical Ada profile
  - Troublesome OOP features are easily avoided
- But no standard annotation facility

Expressibility

- Support for low-level and real-time programming
- Ravenscar Profile for certifiable concurrent applications
- Good inter-language interfacing facilities, to incorporate certifiable libraries from other languages
- Some weaknesses
  - Limited support for distribution / networking

83

Ada Strengths

Sound base language for programming high-reliability systems

Allows construction of tailored high-integrity subsets that are:

- Powerful enough to use for real systems
- Simple enough to be amenable to certification

Scales up to large systems

ISO Standard

- Ada 2005 is the most recent version
- Published by ISO in March 2007

Concurrency support / Ravenscar Profile

Free and publicly available documents

- Ada Reference Manual, Rationale
- Guidelines documents

Good track record in avionics, train control, other safety-critical domains

84

Ada 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

- Order of evaluation in expressions
- Referencing a variable before its initialization

Smaller user / tool vendor community than other languages

85

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 *plus* contracts* *minus* features that interfere with above principle
- Contracts 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

* Sometimes known as “annotations”

86

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)

Contracts

- 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, …
- Control flow analysis – detect dead code, banned constructs (goto, …) } *required*
- Data-flow analysis – detect unused or uninitialized variables, …
- Information-flow analysis – check input / output variable coupling
- Verification condition generation – generate theorems } *optional*
- Theorem proving

87

SPARK and High-Integrity Requirements

Reliability

- Inherits Ada's general advantages (few syntactic surprises)
- Avoids some errors that may occur in full Ada
  - Reference to uninitialized variable

Predictability

- Unambiguous specification with no implementation dependencies
  - But not an official standard

Analyzability

- SPARK Ada subset eliminates features that are complicated to analyze
- Contracts allow statically checkable assertions about resource requirements

Expressibility

- Missing some useful functionality (e.g. generics) but does include some tasking features

88

SPARK Strengths

Designed from the start to satisfy safety-critical (and more generally, high-integrity) requirements

- Language rules are unambiguous, implementation independent
- 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

Contracts show programmer's intent, aid readability

Support for concurrency (Ravenscar profile)

Tedious aspects of constructing proofs are automated

Positive experience across a range of high-integrity projects

- Lower certification costs and post-delivery bug rates

Excellent reference material

- Free and publicly available language definition
- John Barnes' *High Integrity Software* book (see references)

89

SPARK Weaknesses

Language lacks some useful features

- No "pointers", but in many kinds of safety critical systems a restricted style for pointers is permitted
  - Pointers to statically declared data
  - Dynamic allocation at system startup
- Some restrictions (e.g. no recursion) require stylistic workarounds

Mismatch to DO-178B's test-oriented approach

- Formal ("rigorous") methods augment but do not replace coverage analysis etc.

Contracts require different development style than what most programmers are used to

- Attempting to take existing code and "SPARK"ing it is difficult

Relatively small user community

Small number of suppliers for tools, services

90

AdaCore Java - Introduction

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 safety-critical applications

- Generally well-defined semantics for sequential features
- It worries about some safety-related issues
  - Attention to security, bytecode verification
  - Index range checks
  - No unreachable code, 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)
- Some systems being developed in Java may have safety-critical components

91

AdaCore Issues with Java for High-Integrity Systems

“Pure” OO language

- No “free” operation ⇨ system-managed memory reclamation / “Garbage Collection” (GC)
- Garbage collection interferes with predictability and/or adds latency
- Most of the OOT issues identified above arise with Java

Complexity

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

Java environment and execution model clashes with DO-178B

- 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

92

AdaCore Java and High-Integrity 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

93

AdaCore Java and High-Integrity Requirements (2)

Predictability

- Sequential Java is well-defined, except for semantics of finalization
- Thread model is underspecified
- Language definition is not an official standard
- Do not know when Garbage collection occurs

Analyzability

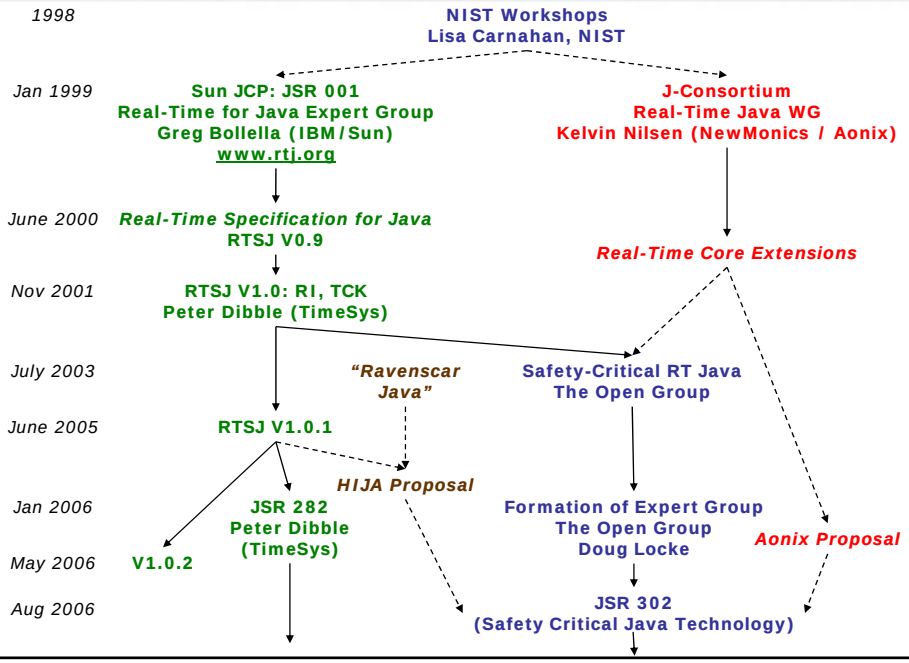
- “Pure” OO language
  - Almost all of the analyzability issues identified above apply, except
    - Interface feature avoids the problems with general multiple inheritance
- Class combines “interface” and “implementation”, complicating analyzabiliy
- Garbage collection issues
- Too complex for safety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

Expressiveness

- Applications not amenable to OOP have a contrived style
- Rich API, but these would need to be certified

94

Background of Real-Time Java Efforts



95

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

Synchronization (priority inversion management)

- Priority inheritance (required), priority ceiling emulation (optional)

96

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

- Specialized kinds of “physical” memory
- “Peek/poke” of primitive data in “raw” memory

97

RTSJ and High-Integrity 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)
  - Java Spec Request work officially started during summer 2006
  - Spec is planned for completion in 2008

98

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

Application Structure

- 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

- No Asynchronous Transfer of Control
- AsynchEventHandlers are included

99

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
- Annotations enable offline (pre-runtime) analysis of resource usage

100

Java Profiles and High-Integrity Requirements

Reliability

- Profiles' restrictions address a few of the issues raised by full Java, but most are intrinsic to the use of Java
- Need adaptation of MISRA-C rules for Java's C syntactic pitfalls

Predictability

- Well-defined semantics
- But 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

- Clumsy to use Java for non-OO processing
- APIs need to be specially written for certifiability

101

C# and Safety/Security

What is C#?

- A "better Java" from Microsoft
- In effect the native language for .NET / Common Language Infrastructure
  - Basis for Common Intermediate Language

Contrast with Java

- Language features
  - Stack-based data structures
  - Better generics
  - Lots of other "bug fixes" and enhancements
- Execution environment
  - Not a virtual machine, but "managed" native code generated either "just in time" or in advance

C# for safety-critical software?

- No current proposals, but the issues would be the same as with Java
  - Same lexical/syntactic structure as in the C family, with many of the same problems
  - Unpredictability stemming from dynamic allocation / garbage collection, underspecified thread semantics

102

AdaCore
The GNAT Pro Company

AdaCore Summary (1)

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

- ☹ Not uniformly enforceable
- ☹ Still based on an intrinsically insecure language
- ☹ Does not readily "scale up" for large systems

C++ "safe subsets"

- ☺ Large potential user community
- ☺ Better than C for large systems

but

- ☹ JSF++ rules not uniformly enforceable
- ☹ No experience yet with MISRA C++
- ☹ Intrinsic issues due to C underpinnings, OOP

103

AdaCore Summary (2)

Ada

- ☺ Firm foundation built on software engineering principles
- ☺ Best fit technically to requirements for High-Integrity OOP
- ☺ Allows user-specifiable subsets
- ☺ Good concurrency model / Ravenscar profile

But:
- ☹ Smaller user / vendor base than other languages

SPARK

- ☺ Expressly designed to support rigorous methods, static analysis
- ☺ Proven track record in high-integrity systems
- ☺ Includes concurrency support

But:
- ☹ Smaller user / vendor base than other languages
- ☹ Restricted feature set

Java: Safety-Critical Java Technology

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

But:
- ☹ Java has intrinsic issues
  - Dynamic / "pure" OO language
- ☹ Work in progress

104

Summary (3) – "Report Card" on Languages

Subjective ratings of language technologies for High-Integrity 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

105

Conclusions on OOT in High-Integrity Systems

There is an essential conflict

- The features that make OOT attractive in general for software development complicate safety and security 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

106

AdaCore
The GNAT Pro Company

Future Directions

Certification standards evolving to account for new language technologies

- Recognition of OOT, other modern features

Security playing an increasing role in safety certification

Languages evolving to address safety/security certification requirements

- Profiles (subsets), support for rigorous methods

Tradeoffs will always exist in choosing a language

- Technical merit – expressibility, consistency with certification requirements
- Cost of certification
- Availability of tools, programmers
- Evaluators' experience with the language technology on previous systems

The distance between "state of the art" and "state of the practice" is shrinking

107

Web Resources and Other References (1)

Safety

- D.S. Herrmann, *Software Safety and Reliability: Techniques, Approaches, and Standards of Key Industrial Sectors*; IEEE Computer Society; 1999
- V. Hilderman & T. Baghai, *Avionics Certification: A Complete Guide to DO-178 (Software) and DO-254 (Hardware)*; Avionics Communications Inc; 2007

DO-178B

- RTCA SC-167 / EUROCAE WG-12. RTCA/DO-178B – *Software Considerations in Airborne Systems and Equipment Certification*, December 1992
- Certification Authority Software Team (CAST): www.faa.gov/aircraft/air_cert/design_approvals/air_software/cast/cast_papers/

DO-178C

- forum.pr.erau.edu/SCAS

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

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

Safety-Critical Systems Club (UK) Tools Directory

- www.scsc.org.uk/tools

108

Web Resources and Other References (2)

Security

- B. Chess & J. West, *Secure Programming with Static Analysis*; Addison Wesley; 2007
- H. Thompson & S. Chase; *The Software Vulnerability Guide*; Charles River Media; 2005
- N. Daswani, C. Kern & A. Kesavan, *Foundations of Security: What Every Programmer Needs to Know*; Apress; 2007
- D. Wheeler, *Secure Programming for Linux and Unix HOWTO*, www.dwheeler.com/secure-programs/
- CERT Secure Coding Standards www.securecoding.cert.org/confluence/display/seccode/CERT+Secure+Coding+Standards
- SANS Top-20 2007 Security Risks: www.sans.org/top20/

Common Criteria

- Common Criteria Portal, www.commoncriteriaportal.org/
- National Information Assurance Partnership / Common Criteria Evaluation and Validation Scheme, www.niap-ccevs.org/
- D.S. Herrmann, *Using the Common Criteria for IT Security Evaluation*, Auerbach Publications, 2003.

109

Web Resources and Other References (3)

Safety and Security

- B. Brosgol, "Safety and Security: Certification Issues and Technologies"; *CrossTalk*, October 2008; www.stsc.hill.af.mil/crosstalk/2008/10/0810Brosgol.html
- C. Taylor, J. Alves-Foss, B. Rinker; *Merging Safety and Assurance: The Process of Dual Certification for Software*; STC 2002
- F. Keblawi & D. Sullivan, "Applying the Common Criteria in Systems Engineering"; *IEEE Security & Privacy*; March/April 2006, pp 50-55
- Common Vulnerability Enumeration: cve.mitre.org/
- Common Weakness Enumeration: cwe.mitre.org/

ISO/IEC JTC 1/SC 22/OWG:Vulnerabilities

- www.aitcnet.org/isai/
- *The intent of the project is to write a document containing guidance to users of programming languages on how to avoid the vulnerabilities that exist in the programming language selected for a particular project.* [From FAQ on website]

Static Analysis Tools

- NIST (National Institute for Standards and Technology); *SAMATE - Software Assurance Metrics And Tool Evaluation*; samate.nist.gov/index.php/Main_Page

110

Web Resources and Other References (4)

C

- R. Seacord, *Secure Coding in C and C++*. Addison-Wesley, 2005

MISRA C

- *MISRA-C: 2004. Guidelines for the use of the C language in critical systems*. 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.
- *MISRA-C++: 2008. Guidelines for the use of the C++ language in critical systems*. www.misra-cpp.org/
- Embedded C++, www.caravan.net/ec2plus/
- S. Dewhurst, *C++ Gotchas: Avoiding Common Problems in Coding and Design*. Addison-Wesley, 2003.
- I. Joyner, *C++?? A Critique of C++ and Programming Language Trends of the 1990s* (3rd edition), 1996.

MISRA C++

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

111

Web Resources and Other References (5)

Ada

- Ada Information Clearinghouse, www.adaic.org
- ISO/IEC TR 15942, *Guide for the Use of Ada in High-Integrity Systems*, 2000. std.dkuug.dk/JTC1/SC22/WG9/n359.pdf
- 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; January 2003
- C. Comar, R. Dewar, and G. Dismukes. "Certification and object orientation: the new Ada answer". In *ERTS 2006*, Toulouse, France, January 2006. (*)

SPARK

- http://www.sparkada.com
- J. Barnes, *High Integrity Software – The SPARK Approach to Safety and Security*, Addison-Wesley, 2003
- M. Croxford & R. Chapman, "Correctness by Construction: A Manifesto for High-Integrity Software", *Crosstalk*, Dec. 2005 www.stsc.hill.af.mil/crossTalk/2005/12/0512CroxfordChapman.html

Formal Methods

- J. Jacky, *The Way of Z: Practical Programming with Formal Methods*; Cambridge University Press; 1997

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

112

AdaCore
The GNAT Pro Company

Web Resources and Other References (6)

Java

- Security Code Guidelines, java.sun.com/security/seccodeguide.html
- Java security hotlist www.cigital.com/javasecurity/hotlist.html

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

Object-Oriented Technology and Safety 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, CA, April 2006 (*)

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

113

Acronyms and Abreviations (1)

API	Application Program Interface
ATM	Air Traffic Management
CAP	Composed Assurance Package (Common Criteria)
CAPP	Controlled Access Protection Profile
CAST	Certification Authority Software Team (DO-178B)
CC	Common Criteria
CCEVS	Common Criteria Evaluation and Validation Scheme
CEM	Common Evaluation Methodology
CNS	Communications, Navigation, and Surveillance
COTS	Commercial Off-The-Shelf
CVE	Common Vulnerability Enumeration
CWE	Common Weakness Enumeration
DO-178B	*[Not an acronym, this is the name/number of a document]*
DER	Designated Engineering Representative (DO-178B)

114

AdaCore
The GNAT Pro Company

Acronyms and Abreviations (2)

EAL	Evaluation Assurance Level (Common Criteria)
EUROCAE	European Organisation for Civil Aviation Equipment
FAA	Federal Aviation Administration (US)
GC	Garbage Collection
HIJA	High Integrity Java
IEC	International Electrotechnical Commission
ISO	International Organization for Standardization
IT	Information Technology
JCP	Java Community Process
JSF	Joint Strike Fighter (US)
JSR	Java Specification Request
JVM	Java Virtual Machine
MC/DC	Modified Condition/Decision Coverage (DO-178B)
MISRA	Motor Industry Software Reliability Association (UK)
NIAP	National Information Assurance Partnership (US)

115

Acronyms and Abreviations (3)

OO	Object Oriented
OOP	Object-Oriented Programming
OOT	Object-Oriented Technology
OOTiA	Object-Oriented Technology in Aviation
OS	Operating System
PP	Protection Profile (Common Criteria)
SAR	Security Assurance Requirement (Common Criteria)
SFR	Security Functional Requirement (Common Criteria)
SIL	Safety Integrity Level
RTCA	*[Not an acronym, this is the name of an organization]*
RTSJ	Real-Time Specification for Java
ST	Security Target (Common Criteria)
TOE	Target of Evaluation (Common Criteria)
TSF	TOE Security Function (Common Criteria)
UML	Unified Modeling Language

116