A Comparison of Ada and Real-Time Java™ for Safety-Critical Applications

Ben Brosgol
brosgol@adacore.com

www.adacore.com

Reliable Software Technologies - Ada Europe 2006
Porto, Portugal
5-9 June 2005
www.ada-europe.org/conference2006.html

Tuesday, 6 June 2006

instituto superior de engenharia do porto

AdaCore
US Headquarters:

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

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

AdaCore
European Headquarters:

8 rue de Milan
75009 Paris France

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

AdaCore

Overview

Summary of issues

Safety certification and language requirements / features

Ada and safety-critical software

Java and safety-critical software

Real-Time Specification for Java (RTSJ) summary

Proposals for safety-critical real-time Java profile

Conclusions

Summary of issues

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, or to the development process

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

- Dilemma: 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

Goal of this presentation

- Assess Ada (including Ada 2005 amendment) and real-time Java for suitability as base languages for safety-critical profiles
  - DO-178B to serve as exemplar of software safety certification standard
  - "Safety critical" = DO-178B, Level A or B

AdaCore

DO-178B and language requirements

Reliability

- No “traps and pitfalls”
- Early error detection
- Compile-time checking

Predictability

- Unambiguous language semantics
- Static demonstration that time / storage constraints satisfied

Analyzability

- Traceability between requirements and code
  - All requirements implemented
  - Each piece of code maps back to a requirement
    - No *dead code* or “Easter eggs”
- Coverage analysis (including *Modified Condition Decision Coverage* at Level A)

Expressiveness

- Support for application-specific real-time functionality (hardware interrupts, etc.)

AdaCore

DO-178B and language features: issues

High-level features [predictability, analyzability]

- Array slice assignment ⇨ implicit loops and conditionals (coverage analysis)
- Access-to-subprogram ⇨ which subprogram is called

Encapsulation [analyzability]

- Information hiding ⇨ coverage analysis

Object-Oriented Programming

- Inheritance [reliability, analyzability]
- Polymorphism [predictability, analyzability]
- Dynamic binding [analyzability]

Generics [analyzability]

Inline expansion [predictability, analyzability]

Run-time support [predictability, analyzability]

- Exception handling, concurrency, memory management
- API

Compiler optimizations [predictability, analyzability]

These issues are addressed in the Object-Oriented Technology in Aviation (“OOTiA”) Handbook, developed under FAA and NASA sponsorship

Ada and safety-critical requirements (1)

AdaCore

Reliability

- Very few features have surprising effects
- Prevention of dangling references to declared objects, subprograms
- Specification of intent on operation inheritance
- Atomic task activation
- Pragma Assert
- But: absence of garbage collection means programmer responsible for memory management

Ada 2005 (applies to: specification of intent on operation inheritance, atomic task activation, pragma Assert)

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

Ada and safety-critical requirements (2)

AdaCore

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
- 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

AdaCore

Java and safety-critical requirements (1)

Why consider Java for safety-critical systems?

- Well-defined semantics for sequential features
- Real-Time Specification for Java (RTSJ) adds deterministic semantics and predictability for the threading features, and non GC'ed memory areas
- Some systems being developed in Java may have safety-critical components

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)
- 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
  - Low-level (error-prone) thread model
  - Absence of named parameter associations

AdaCore

Java and safety-critical requirements (2)

Predictability

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

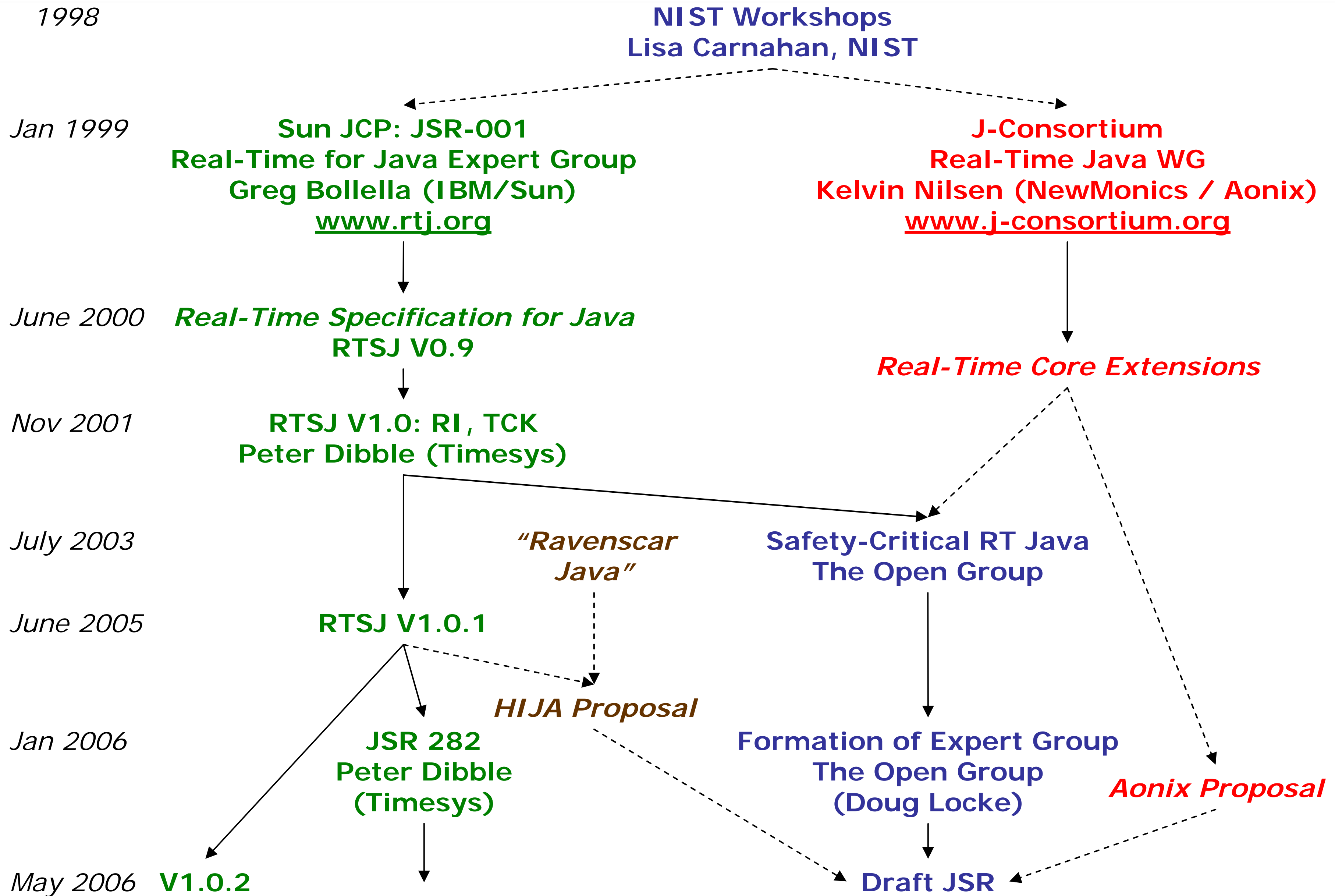
Analyzability

- "Pure" OO language
  - Issues with inheritance, polymorphism, dynamic binding
- 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

Timeline for real-time and safety-critical Java



AdaCore

Real-Time Specification for Java (1)

Goals

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

Concurrency

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

Synchronization (priority inversion management)

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

Real-Time Specification for Java (2)

Memory management

- Garbage-collected heap
- Immortal memory
- Scoped memory areas
  - A scoped area is used for allocations performed by a specified method invocation
  - Reclaimable when no threads reference it
  - Run-time check needed when assigning a reference to a field of an object

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

API

- Use specially implemented standard Java classes

RTSJ and safety-critical requirements

The RTSJ was never intended for safety-critical applications

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

But it does address some of the problems with full Java

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

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

- Started in July 2003 - The Open Group’s Real-Time Embedded Systems Forum
- Two main competing proposals
  - 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)
  - Decision in progress on which proposal to use as starting point
  - Java Spec Request is planned for submission to Sun during summer 2006

AdaCore

HIJA Safety-Critical Java Proposal

General approach

- Basically an RTSJ subset, aimed at certification at DO-178B Levels A and B
- Annotations support static analysis for memory management, synchronization

Concurrency

- RTSJ base scheduler, 28 priorities
- Application comprises periodic and sporadic async event handlers
- No CPU time monitoring, dynamic priorities, on-line feasibility analysis

Synchronization (priority inversion management)

- No synchronized statements, only synchronized methods
- Priority Ceiling Emulation, no blocking ⇨ lockless implementation

Memory management

- Simple uses of scoped and immortal memory, no Garbage Collection

Asynchrony

- No Asynchronous Transfer of Control
- Simplified Async Event handling model

AdaCore

Aonix Safety-Critical Java proposal

General approach

- Inspired by RTSJ, but deviates when deemed necessary
- Annotations support static analysis

Concurrency

- RTSJ base scheduler, 28 priorities
- Only NoHeapRealtimeThreads
- No CPU time monitoring, deadline miss handling, dynamic priorities, on-line feasibility analysis

Synchronization (priority inversion management)

- No synchronized statements, only synchronized methods
- PCP or Atomic (PCP with no blocking)

Memory management

- Immortal memory, "Thread stack" based on static analysis, no Garbage Collection
  - Concern about fragmentation if use RTSJ scoped memory

Asynchrony

- No Asynchronous Transfer of Control
- Simplified Async Event handling model

Java profiles and safety-critical requirements

Reliability

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

Predictability

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

Analyzability

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

Expressibility

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

Conclusions

Ada

- Easier to subset for safety-critical applications
  - OOP can be removed
  - Pragma Restrictions, pragma Profile
- Long history of success in this domain
- Lacks standard annotation facility
  - SPARK offers one approach

Java

- Harder to subset since many issues are intrinsic
  - Error-prone C-based syntax and lexical structure
  - Pure OO language
- Subsetting for static analyzability conflicts with Java's dynamic philosophy
- No experience in producing safety-critical systems in Java
- Java 1.5 annotation facility useful but is too weak
- Work on safety-critical profile will require consensus building

Summary

- Ada is a better technical starting point for safety-critical profiles
- Market interest will keep safety-critical Java an area of active development