

The Ravenscar Profile for High-Integrity Java™ Programs?

Brian Dobbing,
Aonix Europe Ltd,
Partridge House, Newtown Road,
Henley-on-Thames, Oxon RG9 1EN, United Kingdom
brian@aonix.co.uk

Abstract

The Ravenscar Profile was a major output of the 8th International Real-Time Ada Workshop. The profile defines a subset of the Ada95 tasking constructs that matches the requirements of Safety Critical, High Integrity and Hard Real-Time systems by eliminating constructs with high overhead or non-deterministic behavior (semantically or temporally) whilst retaining those elements that form the basic building blocks for constructing analyzable real-time systems.

The recent explosion in interest and use of the Java™ platform for embedded devices has exposed basic flaws in the semantics of its real-time constructs, and this has led to initiatives to extend the language to address these weaknesses. Key personnel involved in the definition of the Real-Time Annex for Ada95 and the Ravenscar Profile also influenced the design of the real-time extensions to Java™, and so it is not surprising that these extensions follow closely the experiences learned from Ada.

This paper describes the proposed set of extensions to Java™ to support real-time and high integrity systems, drawing parallels with the Ravenscar Profile definition. It suggests the possibility of co-existence and interoperability of Ravenscar-compliant Ada and RT-Java partitions within a high integrity or safety critical system.

1 Introduction

Since late 1995, researchers and practitioners have been working on the problem of making the Java language appropriate for development and deployment of real-time applications. The first paper on this topic was published on the web in December of 1995. A subsequent draft of that paper is cited as reference [1].

One of the obstacles hindering quick adoption of real-time Java technologies has been the absence of standards. Starting in June of 1998, under the auspices of the National Institute of Standards and Technology (NIST), a

number of companies began meetings to develop consensus-based requirements for real-time extensions for the Java platform. The culmination of this group's efforts, which ultimately included participation of 38 different entities, was a document titled "Requirements for Real-Time Extensions for the Java Platform", published in September 1999 [2].

The NIST-sponsored effort focused on defining requirements for Java real-time extensions. That group made a conscious choice not to develop a specification for real-time Java extensions.

Two independent groups have undertaken the task of defining specifications for standard real-time extensions for the Java platform based on the NIST requirements document. The Real-Time Java Working Group, hosted by the J Consortium, was formed in November 1998. This group, which follows an open consensus-based process patterned after the processes established by existing ISO Publicly Available Specification (PAS) submitters, released its draft specification for a 45-day public review and comment period beginning on September 27, 1999 [3]. On Feb. 3, 2000, the Real-Time Java Working Group published a revision of the original specification, hereafter known as the "Real-Time Core". The other group, formed by Sun Microsystems in March of 1999, is known as the "Real-Time Expert Group". Their specification has also recently completed its public review (see reference [4]).

The NIST requirements recommend that the Real-Time extensions should not attempt to address the additional requirements imposed by specialized market sectors such as high integrity and distributed systems. Rather, the meeting of such requirements should be addressed by the use of *profiles*, where a profile can either add or remove APIs relative to the specification. Consequently several working groups were set up within the J Consortium to define such profiles for the Real-Time Core. This paper outlines the Real-Time Core specification and presents the subset that is suitable for inclusion in the High Integrity profile.

The Ada95 revision process underwent similar evolution. Ada83 contained serious drawbacks that impeded its wide acceptance by the real-time community that were addressed by the Ada95 standard. A goal of this standard was to address the requirements of specific market sectors by the use of optional *Annexes*, and the Real-Time Annex definition introduced functionality to provide deterministic temporal operation (see reference [5]). In 1997, the *Ravenscar Profile* was a major output of the 8th International Real-Time Ada Workshop, and is described in reference [6]. This profile defines a subset of the Ada95 tasking constructs that matches the requirements of Safety Critical, High Integrity and Hard Real-Time systems by eliminating constructs with high overhead or non-deterministic behavior (semantically or temporally) whilst retaining those elements that form the basic building blocks for constructing analyzable real-time systems.

Thus very clear parallels can be established between the Real-Time Core and the Ada95 Real-Time Annex, and between the High Integrity Profile and the Ravenscar Profile.

2 Real-Time Core Specification

The J Consortium specification [3] addresses all of the core requirements identified in the NIST requirements document along with a number of additional self-imposed requirements that were agreed upon by the members of the J Consortium's Real-Time Java Working Group.

In summary, the NIST requirements specify that the real-time extensions should:

- provide services comparable to the services offered by commercially available real-time operating systems
- be reasonably straightforward to implement
- provide a foundation upon which more sophisticated higher level real-time profiles can be constructed
- not endeavor to advance the state of the art in real-time programming

Space does not allow for a complete description and analysis of the core NIST requirements. This topic is treated in Section 5 of reference [2].

The NIST requirements leave considerable room for interpretation and ambiguity. In order to establish a foundation upon which the Real-Time Java Working Group's draft core specification would be built, the Real-Time Java Working Group established a number of clarifying principles to augment the list of core requirements identified in the NIST requirements document. These working principles follow. For purposes of this discussion, the term "*Baseline Java*" refers to the 1.1 version of the Java language, as it has been defined by Sun Microsystems, Inc.

- The Core Java system must support limited cooperation with Baseline Java programs running on the same Java virtual machine, with the integration designed so that neither environment needs to degrade the performance of the other.
- The Core Java system must support limited cooperation with programs written according to the specifications for higher level real-time Java profiles in environments that implement these optional real-time profiles, with the integration designed so that neither environment needs to degrade the performance of the other.
- The semantics of the real-time core shall be sufficiently simple that interrupt handling latencies and context switching overheads for Core Java programs can match the latencies and context switching overheads of today's RTOS products running programs written in C, C++ and Ada.
- The Core Java specification shall enable implementations that offer throughputs that are essentially the same as are offered by today's optimizing C++ compilers, except for semantic differences required, for example, to check array subscripts.
- Core Java programs need not incur the run-time overhead of coordinating with a garbage collector.
- Baseline Java components and components written for yet-to-be-defined higher-level real-time profiles shall be able to read and write the data fields of objects that reside in the Core Java object space.
- Security mechanisms will prevent Baseline Java components from compromising the reliability of Core Java components.
- Core Java programs shall run on a wide variety of different operating systems, with different underlying CPUs, and integrated with different supporting Baseline Java virtual machines. There shall be a standard way for Baseline Java components to load and execute Core Java components.
- The Core Java execution environment shall exist in two forms, one supporting dynamic loading and unloading of core classes and the other not supporting dynamic class loading.
- The Core Java specification shall support the ability to perform stack allocation of dynamic objects under programmer control.
- The Core Java specification shall be designed to support a small footprint, requiring no more than 100K for a typical Static Core Java Execution Environment.
- The Core Java specification shall enable the creation of profiles which expand or subtract from the capabilities of the Core Java foundation.
- All Core Java classes shall be fully resolved and initialized at the time they are loaded.

2.1 Key Aspects of the Core Extensions

For a complete characterization of the Core Java real-time extensions, see reference [3]. In overview, the Core Java Execution Environment:

- is a plug-in module that can augment any Java virtual machine. This allows users of the Core Java Execution Environment to leverage the large technology investment in current virtual machine implementations, including byte-code verifiers, garbage collectors, JustInTime compilers, and symbolic debuggers
- can be configured to run without a Baseline Java virtual machine. This allows developers of Core Java applications to deploy in very small memory footprints (as small as 100 Kbytes)
- does not support automatic garbage collection. Core Java programmers are required to explicitly release objects when they are done using them
- supports stack allocation of objects, under programmer control
- supports priority inheritance for synchronization and explicit lock objects
- uses the Priority Ceiling Protocol (PCP) for specially designated objects
- supports signaling and counting semaphores, direct access to I/O ports, and the implementation of interrupt handlers as Core Java components
- allows asynchronous events to trigger control transfer in particular tasks. Each task has the opportunity to define the response to an asynchronously signaled event to either handle the event and then resume whatever work was preempted, or to abort its current efforts.

2.2 Similarity with Ada95 Real-Time Annex

Almost all new elements in the Core Java extensions, and indeed the parallel extensions defined by the Sun Microsystems Experts Group specification, can be found in either the core language definition for Ada95 or its Systems Programming or Real-Time Annex [5]. These include:

- A large range of priority values (128 values in the Core Java specification)
- Well-defined thread scheduling based on FIFO_Within_Priorities policy
- Addition of Protected Objects to the existing Synchronized objects and methods, that prohibit voluntary suspension operations and that define a Ceiling Priority for implementation of mutual exclusion (c.f. Ceiling_Locking policy)
- Addition of asynchronous event handlers (c.f. protected entries)

- Addition of asynchronous transfer of control triggered by either time expiry or an asynchronous event
- Allocation and access of objects at fixed physical memory locations, or in the current stack frame
- Suspend / Resume primitives for threads (c.f. suspension objects)
- Dynamic priority change for threads
- Absolute time delay (c.f. delay_until statement)
- Use of nanosecond precision in timing operations (c.f. Ada.Real_Time)
- Definition of interrupt handlers and operations for static and dynamic attachment

Thus the real time extensions for Java are quite compatible with the Ada95 Real-Time Annex execution model, which encourages the view that both languages could be used to develop parallel subsystems that execute over a common RTOS. Indeed the compatibility extends also to the *generic* POSIX “pthreads” specification that is supported in various flavors by many of the commercial RTOS systems that are in use today.

2.3 Dissimilarity with Ada95 RT Annex

The following design decisions were taken during the development of the Real-Time Core that conflict with those taken for Ada95:

- Low-level POSIX-like synchronization primitives, such as mutexes and signaling and counting semaphores, are included as well as the higher-level of abstraction provided by synchronized objects (mutual exclusion regions), monitors and protected objects. Ada95 chose to provide only the higher level of abstraction such as the protected object and the suspension object. There is therefore greater scope for application error using the Real-Time Core, such as accidentally leaving a mutex locked.
- More than one locking policy is present. Synchronized objects and semaphores require only mutual exclusion properties and so are subject to priority inversion problems. Mutex locks and monitors require priority inheritance to be applied in addition to mutual exclusion. Protected objects require instead the priority ceiling protocol to be applied as for Ceiling_Locking in Ada95. The requirement on the underlying RTOS to support both inheritance and ceiling locking was one that Ada95 chose not to impose. Also the introduction of protected objects with Ceiling_Locking in Ada95 has implicitly deprecated the Ada83 rendezvous that was prone to priority inversion problems.
- The only mutual exclusion region that is abort-deferred is the Atomic interface used by interrupt handlers. In particular, protected object and monitor operations are not abort-deferred regions. This removes the integrity guarantees that a designer may well be relying on in a

protected object. Use of the Atomic interface introduces a number of coding restrictions that limit its general applicability (in particular all the code must be execution-time analyzable) and so this may not be appropriate in all scenarios. In Ada95, all protected operations are abort-deferred and there is no restriction on the content of the code other than that it does not voluntarily suspend.

- There is no notion of requeue in the Real-Time Core. Ada95 requeue has been found to be useful in designing scenarios such as servers that provide multi-step service.
- Asynchronous transfer of control includes the ability to resume execution at the point of interruption (i.e. effectively discarding the interrupt) which could be useful for example to ignore an execution time overrun signal in certain context-specific situations.

3 High Integrity Profile

The Ravenscar Profile [6] was created specifically to support basic Ada95 concurrency constructs to meet the requirements of hard real-time, high integrity and safety critical systems. Thus if real-time Java is going to be able to be used in the same market sectors, it is necessary to create a similar subset of the Core Java specification concurrency features, in addition to addressing issues of determinism and fault tolerance in its sequential execution model. This work is in progress within the J-Consortium in the form of the High Integrity Profile working group. The high integrity profile is designed to meet the requirements of:

- Safety Critical / High Integrity, for which all the data and executable code must undergo thorough analysis and testing, and must be guaranteed not to be corruptible by less trusted code
- High Reliability / Fault Tolerance, for which the code must be resilient enough to detect faults and to recover with minimal disturbance to the overall system
- Hard Real-Time, for which the timing of code execution must be deterministic to ensure that deadlines are met
- Embedded, for which a small footprint and fast execution are required.

These requirements map into the following generic functionality:

- Partitioning that allows for hardware protection to be applied to critical code and data to preserve its integrity, and for the creation of standby “replicas” to take over if a fault occurs
- Deterministic memory management that ensures that the memory requirements of the system can never become exhausted

- Deterministic code execution that supports timing analysis and the maintenance of replicas in step with one another
- Exception catching to detect faults
- Access to special physical memory areas to checkpoint program state for deterministic error recovery
- Elimination of complex constructs to reduce the footprint, increase execution speed, and simplify the verification of the runtime system required for formal certification.

3.1 Partitioning

Partitioning is an essential component of systems that contain a mixture of critical and non-critical software. The critical code must be protected by a spatial firewall that ensures absence of intrusion by less-trusted code. Such firewalls are often enforced by hardware when a memory management unit is available. This implies that the high integrity execution model supports only the Static Core Java Execution Environment, and so does not make use of the “plug-into-the-VM” feature of the Real-Time Core. Consequently there is no exchange of objects between the VM and the high integrity program, nor are any of the VM features used, such as dynamic loading. The high integrity partition is statically-linked native code that executes in protected memory space. All communication into and out of the high integrity partition must be via an underlying kernel, for example using the I/O ports capability of the Real-Time Core. Dynamic update is defined at the partition-level only, rather than at the class level. Thus a replacement partition can be loaded and activated in order to achieve dynamic functional extension or error correction in the application.

In addition to the spatial firewall, the high integrity partition requires a temporal firewall to exist to ensure that it meets all of its critical deadlines. This firewall must be enforced by the underlying kernel which, by implication, is also high integrity code. The kernel can elect to perform fixed time-slice round-robin scheduling of all partitions, or to implement a partition priority scheme such that the high integrity partition gets as much CPU time as it needs.

3.2 Memory Management

The use of dynamic memory management in the high integrity partition must be deterministic. In common with the Real-Time Core, automatic garbage collection is not supported, and dynamic memory usage is required not to fragment.

This is achieved by allocating each object according to its lifetime based on one of the following:

- Stack allocation of method-local objects under programmer control that are automatically reclaimed when the enclosing method exits.
- Use of per-thread fixed-size “allocation contexts” for thread-local objects that are automatically re-used on each invocation of the thread’s run method.
- Global allocation at initialization time in a special allocation context for immortal objects.

Thus the high integrity partition makes no use of a general *heap*, with all its associated non-deterministic compaction and garbage collection requirements.

3.3 Concurrency

The high integrity profile supports the full range of priorities and the three extensions from the basic CoreTask class that are defined in the Core Java specification:

- A Periodic task is automatically made ready to run by the underlying kernel at the start of each period
- A Sporadic task is automatically made ready to run by the underlying kernel whenever its associated asynchronous event object is fired
- An Interrupt task is automatically made ready to run by the underlying kernel whenever the interrupt to which it is attached is delivered

In addition, the profile supports the basic CoreTask class whose run method must be explicitly started by another thread.

All threads are created at program startup, for example as part of the initialization code for the classes. There is no support for declaring a thread class as an inner class within say a method, and so no requirement for any implicit *join* interface to prevent a thread from accessing the local objects of its enclosing method after that method has exited.

Thread operations also include the ability to suspend until an absolute clock time is reached and to suspend on a signaling semaphore until resumed by another thread.

Contention for access to shared resources is achieved via *protected objects*. With regard to deterministic temporal behavior, it is necessary to compute statically the worst case blocking time for a thread when accessing a shared resource. Current schedulability analysis theory indicates that use of the *Priority Ceiling Protocol* (PCP) provides the most optimal worst case blocking times, and so PCP is mandated by the high integrity profile as the only supported locking mechanism for protected objects.

The Core Java specification defines priority inheritance as applying to:

- Mutual exclusion locks (mutexes)
- Synchronized methods (monitors)

Since the co-existence of priority inheritance and PCP adds significant complexity both to timing analysis and to the runtime implementation, these constructs are excluded from the high integrity profile.

The Core Java specification defines mutual exclusion also for:

- Synchronized objects
- Counting semaphores

Since these constructs undermine schedulability analysis as a potential source of unbounded priority inversion, they are excluded from the high-integrity profile.

Core Java tasking constructs that define asynchronous thread-to-thread operations are also excluded from the high integrity profile in order to ensure predictability of execution. These include:

- Stop (abort)
- SetPriority
- Suspend (asynchronous) / Resume
- Asynchronous Transfer of Control (ATC) based on events

The elimination of these constructs implies that protected operations do not require support for abort-deferred regions in the high-integrity profile, nor priority nor suspension deferral. This simplifies the implementation as well providing the necessary deterministic operation of the threads in general.

3.4 Error Recovery

In order to detect and to recover from faults at runtime, the high integrity profile supports exception *throw* and *catch* clauses. In addition, the profile supports access to specific physical addresses to allow objects to be mapped, for example, in persistent memory to save state for recovery purposes. This would also allow access to fast memory to copy state from one partition to its replica, in order to keep standbys “warm”, or to do fast state comparison to verify that a standby is “hot”.

3.5 Similarity with Ravenscar Profile

The Ravenscar Profile definition has provided a framework for the High Integrity profile specification and so it is not surprising that there is a high degree of compatibility. All threads are global (there is no dependency hierarchy). There is support for periodic threads whose release is based on the real-time clock; aperiodic threads whose release is based on asynchronous events or interrupts, and shared data protected by PCP-based protected objects. There is no requeue, abort, dynamic priority change, asynchronous suspension or ATC. Scheduling is defined as FIFO_Within_Priorities.

With regard to non-concurrency-related topics, there is no general heap in the high integrity profile though something similar to Ada95 Storage Pools is supported via *allocation contexts*.

Java exception throwing and catching is supported, though all exceptions are pre-allocated, and so exception

objects do not contain state information. This is analogous to supporting Ada83-style exceptions rather than Ada95 exception occurrences.

4 Conclusion

The Core Java specification has advanced Java into the forefront of languages and environments suitable for implementing real-time embedded systems. The specification is compatible with current POSIX-compliant RTOS systems, although it contains some weaknesses and minor incompatibilities with the Ada95 execution model.

The High Integrity Profile further advances Java into critical systems whose verification requirements are totally alien to those addressed by Baseline Java, and which have been implemented to date only by first-class language environments such as provided by Ada95. The High Integrity Profile is totally compatible with the execution model prescribed by the Ada95 Ravenscar Profile, and so co-existence of partitions built with these profiles is feasible even within applications with the most stringent safety requirements, thereby maximizing the benefits of the strengths of each language.

5 References

- [1] Kelvin Nilsen. *Issues in the Design and Implementation of Real-Time Java*. July 1996. Published June 1996 in “Java Developers Journal”, republished in Q1 1998 “Real-Time Magazine”. <http://www.newmonics.com/pdf/RTJI.pdf>.
- [2] Lisa Carnahan and Marcus Ruark, editors. *Requirements for Real-Time Extensions for the Java Platform*. Published by National Institute of Standards and Technology. September 1999. Available at <http://www.nist.gov/rt-java>.
- [3] J Consortium, *Real-Time Core Extensions for the Java Platform*, Draft 1.0.10, February 3, 2000. Available at <http://www.j-consortium.org>.
- [4] The Real-Time Expert Group, *The Real-Time Specification for Java*, Draft 0.9.1, Feb. 3, 2000. Available at <http://www.rti.org>.
- [5] *Ada95 Reference Manual*, International Standard ANSI/ISO/IEC-8652:1995, January 1995
- [6] B.J.Dobbing and A.Burns, *The Ravenscar Tasking Profile for High Integrity Real Time Programs*. In *Reliable Software Technologies – Ada-Europe ’98*, Lecture Notes in Computer Science 1411, Springer Verlag (June 1998)