Real-Time Convergence of Ada and Java™

Ben Brosgol
Ada Core Technologies,
79 Tobey Road,
Belmont, MA 02478
United States of America
brosgol@gnat.com

Brian Dobbing
Praxis Critical Systems,
20 Manvers Street,
Bath BA1 1PX
United Kingdom
brian@praxis-cs.co.uk

1 ABSTRACT

Two independent recent efforts have defined extensions to the Java platform that intend to satisfy real-time requirements. This paper summarizes the major features of these efforts, compares them to each other and to Ada 95's Real-Time Annex, and argues that their convergence with Ada95 may serve to complement rather than compete with Ada in the real-time domain.

1.1 Keywords

Real-Time, Ada, Java, threads, scheduling, garbage collection, asynchrony

2 INTRODUCTION

Over the past several years the computing community has been coming to grips with the Java platform, a technology triad comprising a relatively simple Object-Oriented Language, an extensive and continually growing set of class libraries, and a virtual machine architecture and class file format that provide portability at the binary level. Java was first introduced as a technology that could be safely exploited on "client" machines on the Internet, with various levels of protection against malicious or mischievous applets. However, as interest in Java's promise of "write once, run anywhere" has increased, the platform's application domain has been expanding dramatically.

One area that has been attracting attention is real-time systems. On the one hand, that should not be completely surprising. The research project at Sun Microsystems that spawned Java was attempting to design a technology for embedded systems in home appliances, and embedded systems typically have real-time constraints. Moreover, Java is more secure than C and simpler than C++, and it has found a receptive audience in users dissatisfied with these languages. And unlike C and C++, Java has a built-in model for concurrency (threads) with low-level "building blocks" for mutual exclusion and communication that seem to offer flexibility in the design of multi-threaded programs. Parts of the Java API address some real-time application areas (for example `javax.comm` for manipulating serial and parallel devices), and V1.3 of the Java Software Development Kit has introduced a couple of utility classes for timed events. Java therefore may seem a viable candidate for real-time systems, especially to an organization that has adapted Java as an enterprise language.

However, even a casual inspection of Java reveals a number of obstacles that interfere with real-time programming. In this introductory section we will summarize these issues and then briefly describe how they have been addressed.

2.1 Challenges

The main problems for Java as a real-time technology fall into several areas, mostly related to predictability.

Thread model

Although Java semantics are consistently deterministic for the sequential parts of the language (e.g. the order of expression evaluation is defined as left-to-right, references to uninitialized variables are prevented) they are largely implementation-dependent for thread scheduling. The Java Language Specification explicitly states [JLS00, Section 17.12]:

SIGAda 2001 09/01 Bloomington, MN, USA

> "*... threads with higher priority are generally executed in preference to threads with lower priority. Such preference is not, however, a guarantee that the highest priority thread will always be running, and thread priorities cannot be used to reliably implement mutual exclusion.*"

This flexibility makes it impossible to ensure that real-time threads will meet their deadlines. The implementation may or may not use priority as the criterion for choosing a thread to make ready when a lock is released. Even if it did, unbounded priority inversions could still occur since there is no requirement for the implementation to provide priority inversion avoidance policies such as priority inheritance or priority ceiling emulation. There is also no guarantee that priority is used for selecting which thread is awakened by a `notify()`, or which thread awakened by a `notifyAll()` is selected to run.

Other facets of the thread model also interfere with real-time requirements. The priority range (1 through 10) is too narrow, and the relative `sleep()` method is not sufficient: the standard idiom for simulating periodicity with this method can lead to a missed deadline if the thread is preempted after computing the relative delay but before being suspended.

A more detailed analysis of Java threads, with a comparison to Ada's tasking model, may be found in [Brosgol98].

Memory management

Despite its C-like syntax, Java belongs semantically to a family of Object Oriented languages including Simula, Smalltalk, and Eiffel: languages that provide no mechanism for programmers to reclaim storage but which instead are implemented with automatic memory reclamation ("Garbage Collection" or "GC"). The idea of garbage collection in a real-time program may sound like a contradiction in terms, but there have been a number of incremental and concurrent collectors that attempt to address the predictability problems of a classical mark-and-sweep strategy [Jones97]. Nevertheless, efficient real-time garbage collection is still more a research topic than a mainstream technology. This is a particular issue for Java, since all objects (including arrays) go on the heap.

Dynamic Semantics

One of the main attractions of Java is its run-time flexibility. For example, classes are loaded dynamically, intraspection allows the run-time interrogation of a class's properties, and cutting-edge compiler technology allows optimized code to be generated during program execution. Unfortunately, all of these capabilities conflict with the traditional static environment ("compile, download, run") for real-time programs. Implementing Java with static linking is possible but difficult, and necessitates restrictions on the use of certain language features.

Asynchrony

A real-time program typically needs to respond to asynchronous events generated by either hardware or software, and sometimes needs to undergo asynchronous transfer of control ("ATC"), for example to time out if an operation is taking too long. The Java Beans and AWT event registration and listener model is a reasonable framework for asynchronous events but omits semantic details critical to real-time programs, such as the scheduling of event handlers. The `interrupt()` method requires polling and thus is not an ATC mechanism. The methods related to ATC have either been deprecated (`stop`, `suspend`, `resume`) or are discouraged because of their proneness to error (`destroy`). Thus Java is rather weak in the area of asynchrony.

Object-Oriented Programming

OOP support is one of Java's most highly touted strengths, but the real-time community has traditionally been very conservative in its programming style and still views OOP with some skepticism. The dynamic nature of OOP (for example the dynamic binding of instance methods) interferes with static analyzability, and Garbage Collection introduces unpredictability or high latency.

Application Program Interface

Class libraries that are to be used in real-time programs need to be implemented specially in order to ensure that their execution time be predictable. This is partly a programming issue (e.g. choice of algorithms and data structures) and partly a JVM implementation issue (Garbage Collection strategy).

Missing functionality

With the goal of language simplicity, the Java designers intentionally omitted a number of features that might be useful in a real-time program, such as general unsigned integral types, strongly-typed scalars, and enumeration types. Other omissions impinge on programming in general, such as generic templates and operator symbol overloading. The language and API also lack system programming facilities for accessing the underlying hardware (such as "peek" and "poke" to access numeric data at physical addresses).

Performance

Although "real time" does not mean "real fast", run-time performance cannot be ignored. Java has several challenges in this area. The key to "write once, run anywhere" is the JVM and the binary portability of class files. But a software interpreter introduces overhead, and hardware implementations are not mainstream technology. Garbage Collection and the absence of stack-

resident objects have an obvious performance impact, and there is also the problem that array initializers result in run-time bytecodes to be performed, versus having a ROMable image in the class file.

2.2 The NIST Requirements and the Two Real-Time Java Efforts

The problems with Java as a real-time technology are steep but not insurmountable. Given the potential market and the fascinating technical issues it is not surprising that real-time Java has been a topic of active investigation. Probably the earliest work was by Kelvin Nilsen in late 1995 and early 1996 [Nilsen96]. Subsequently Lisa Carnahan from the National Institute for Standards and Technology in the U.S. (NIST) took the lead in organizing a series of workshops to identify the issues and to develop consensus-based requirements for real-time extensions to the Java platform. The culmination of this group's efforts, which ultimately included participation by 38 different entities, was a document titled "Requirements for Real-Time Extensions for the Java Platform", published in September 1999 [NIST99].

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

Two independent groups have undertaken to define such specifications. One is the Real-Time Java Working Group under Kelvin Nilsen from Newmonics; this group was formed in November 1998 and is under the auspices of a set of companies and individuals known as the J-Consortium. The other effort is from the Real-Time for Java Expert Group ("RTJEG") under Greg Bollella (then with IBM, now at Sun Microsystems). The RTJEG was established under the terms of Sun's Java Community Process; the product of this effort is a specification, a reference implementation, and a conformance test suite. As of early 2001, the first draft of the specification has been published [Bollella00], while the reference implementation and the test suite formulation are in progress.

The split into two efforts versus a single undertaking was motivated by business considerations rather than technical factors. Participation in the RTJEG required signing an agreement with Sun Microsystems that some organizations found problematic. However, the two efforts ended up taking technical approaches that are more complementary than duplicative. As will be seen below when we cover the specifications in more detail, the J-Consortium has focused on defining real-time "core" kernel facilities external to a JVM, whereas the RTJEG has defined an API that needs to be supported within a JVM implementation. Indeed the J-Consortium specification can be regarded as providing a set of kernel services as might be found in an RTOS for Java.

3 REAL-TIME CORE SPECIFICATION

3.1 Summary

In order to establish a foundation upon which its Real-Time Core Extensions to the Java platform specification [JCons00] would be built, the J Consortium's Real-Time Java Working Group (RTJWG) established a number of clarifying principles to augment the full list of key requirements identified in the NIST requirements document for real-time extensions to Java [NIST99]. 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, and "Core Java" refers to an implementation of the Real-Time Core Specification.

The Core Java execution environment shall exist in two forms: the *dynamic* one that is integrated with a Java virtual machine and supports dynamic loading and unloading of Core classes, and the *static* one that is stand-alone and does not supporting dynamic class loading.

The Core Java dynamic execution environment shall 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 specification shall define distinct class hierarchies from those defined by Baseline Java.

The Core Java specification shall enable the creation of *profiles* which expand or subtract from the capabilities of the Core Java foundation.

The Core Java system shall support limited cooperation with programs written according to the specifications of these *profiles*, with the integration designed so that neither environment needs to degrade the performance of the other.

The semantics of the Core Java specification 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 comparable to those 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 according to the specifications of profiles, shall be able to read and write the data fields of objects that reside in the Core Java *object space*.

Security mechanisms shall prevent Baseline Java and other external profile components from compromising the reliability of Core Java components.

Core Java programs shall be runnable 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 specification shall support the ability to perform memory management of dynamic objects under programmer control.

The Core Java specification shall support a deterministic concurrency and synchronization model with features comparable to those in Real-Time Operating Systems.

The Core Java specification shall be designed to support a small footprint, requiring no more than 100K bytes for a typical static Core Java execution environment.

All Core Java classes shall be fully resolved and initialized at the time they are loaded.

In summary, the Core Java execution environment:

either is a plug-in module that can augment any Baseline 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, dynamic loaders, and symbolic debuggers;

or can be configured to run without a Baseline Java virtual machine. This allows users of the Core Java execution environment to develop high performance kernels deployed in very small memory footprints.

3.2 Concurrency and Synchronization

3.2.1 Scheduling and Priorities

The Core Java specification supports a large range of priorities. Each implementation is required to support a minimum of 128 distinct values, with the highest N being used as interrupt priorities, where N is implementation-defined. In addition, the Core Java semantics require preemptive priority-based scheduling as defined by the *FIFO_Within_Priorities* policy. Support for time-slicing is also defined but not required. This model is in marked contrast to Baseline Java's small priority range (10 values) and absence of guarantee that a higher priority task will preempt a low priority task when it is ready to run. Alternative scheduling policies may be specified via profiles.

The Core task class hierarchy is rooted at `CoreTask`. A *CoreTask* object must be explicitly started via the `start()` method. There are two specialized extensions of *CoreTask*:

The `SporadicTask` class defines tasks that are readied by the occurrence of an *event* that is triggered either periodically or via an explicit call to its `fire()` method.

The `InterruptTask` class defines tasks that are readied by the occurrence of an *interrupt* event, making them analogous to interrupt service routines.

3.2.2 Task Synchronization Primitives

Task synchronization is provided in the Core Java specification via a number of different features, a first group of which supports priority inversion avoidance and a second group of which does not.

In the first group, Baseline Java-style usage of ***synchronized*** methods and ***synchronized(this)*** constructs are both supported and define transitive priority inheritance to limit the effects of priority inversion. In addition, traditional POSIX-style *mutexes* are supported, and these also define transitive priority inheritance to be applied when there is contention for the lock. Finally there is support for Ada-style *protected objects* that are locked using priority ceiling protocol (actually, the same emulation using immediate ceiling locking that is defined in Ada95), and that prohibit execution of suspending operations, e.g. `wait()`. An extension of the priority ceiling protocol interface, known as the `Atomic` interface, is defined for *InterruptTask* `work()` methods. This interface implies that all the code (at the bytecode level) must be statically execution-time-analyzable. The intent is to be able to guarantee the static worst case execution time bounds on interrupt handler execution.

In the second group, POSIX-style counting and signaling semaphores are supported. There is no concept of a single owner of a semaphore and hence there is no priority inheritance on contention (and unbounded priority inversion may result). A counting semaphore may have *count* concurrent owners. A signaling semaphore is similar to Ada's *suspension object* except that multiple

tasks can be waiting for the signal. A signal that occurs when there are no waiting tasks is a no-op.

3.3 Memory Management

3.3.1 Core Object Space

The Core Java requirements include the provision of security measures to ensure that the Baseline Java domain cannot compromise the integrity of Core objects in the dynamic Core execution environment. This is realized in the specification by segregating Core objects into their own object space that is quite separate from the Baseline Java heap.

However another requirement of the dynamic Core Java execution environment is to provide limited and controlled communication with the Baseline Java domain. This is achieved via the `CoreRegistry` class that includes a `publish()` method to publish the names of externally-visible Core objects. A `lookup()` method is also provided for the Baseline Java domain to obtain a reference to the published Core object. However the Baseline Java domain is only permitted to call special *core-baseline* methods that are explicitly identified within the published object, and so access to the Core object space is totally defined and controlled by the operation of these methods.

3.3.2 Garbage Collection

A key requirement of the Core Java specification is that the system need not incur the overhead of traditional automatic garbage collection. This is intended to provide the necessary performance and predictability, avoiding overheads such as read/write barriers, object relocation due to compaction, stack/object scanning and object description tables, as well as avoiding the determinism problems associated with executing the garbage collector thread.

3.3.3 Core Object Allocation / Deallocation

A garbage collector is an essential component of a Baseline Java VM due to Java's object lifetime model, which does not provide an explicit deallocation operation, nor do the semantics provide known points for an implementation to perform guaranteed implicit deallocation. Hence the Core specification defines an alternative strategy for memory allocation and reclamation under programmer control. This is achieved via the following:

An object whose reference is declared locally within a method can be explicitly identified as *stackable*, which means that the object lifetime is asserted to be no greater than that of the enclosing method. Various restrictions are defined for stackable objects to prevent dangling references. Thus an implementation may allocate stackable objects on the runtime method stack in the same way as Ada implementations of local variables.

A class called *AllocationContext* is defined in the Core specification that is somewhat analogous to Ada's *Root_Storage_Pool* type in that it provides a facility for declaring an area for dynamic memory allocation (including at a specific memory address) and the means to control it programmatically. Each task automatically implicitly allocates an allocation context upon creation, and this storage area is used by default for allocation of its non-stackable objects.

The base *AllocationContext* class provides a `release()` method that reclaims all the objects in the context in an unchecked way that could lead to dangling references as for Ada's `Unchecked_Deallocation`. The model of allocation contexts allows an application to implement various paradigms such as *mark/release* heaps, and *factory* methods that construct objects in a known storage area. In addition, all objects (with one exception) that are allocated in implicit task-specific allocation contexts are automatically reclaimed when the task terminates either normally or via an external call to its `stop()` method. The exceptional case is for objects (and those they reference) that are published to the Baseline Java domain. This ensures that the Baseline Java domain cannot get a dangling reference from the use of a published Core Java object. An implementation of the dynamic Core Java execution environment is therefore required to detect when published objects are no longer accessible by the Baseline Java domain such that they can become candidates for automatic reclamation at task termination. Stackable objects are not permitted to be published to Baseline Java.

3.4 Asynchrony

Asynchronous interaction between Core tasks is achieved in the Core Java specification via *events*. There is also the `abort()` method to kill a task. Two event models are defined:

a) the firing and handling of asynchronous events

b) asynchronous transfer of control.

3.4.1 Asynchronous Events

Three kinds of asynchronous event are defined by the Core Java specification:

PeriodicEvent is defined to support periodic tasks. The event fires at the start of each period which causes the associated periodic event handler task to become ready to execute its `work()` method.

SporadicEvent is defined to support sporadic tasks that are triggered by software. The event is explicitly fired by a task which causes the associated sporadic event handler task to become ready to execute its `work()` method

InterruptEvent is defined to support interrupt handling. The event can be explicitly fired by a task (to achieve a software interrupt) or implicitly fired by a hardware interrupt. This causes the associated interrupt event handler task to become ready to execute its `work()` method, which must implement the *Atomic* interface as described in section 3.2.2.

3.4.2 Asynchronous Transfer of Control

Asynchronous transfer of control is supported in the Core Java specification by building upon the asynchronous event model. A special `ATCEvent` class is defined for the event itself. Each task construction may specify a handler for *ATCEvent* that is invoked whenever another task calls the `signalAsync()` method, unless the task is currently in an abort-deferred region. If the handler returns normally, the *ATCEvent* is handled without causing a transfer of control, i.e. the task resumes at the point at which it was interrupted. This is useful in situations where the *ATCEvent* is to have minimal or no effect, such as ignoring a missed soft deadline.

Otherwise, if the handler returns by raising a special `ScopedException` object and the task is in an ATC-enabled execution scope, then the exception causes the transfer of control in the task. An ATC-enabled scope is created by constructing a *ScopedException* object and having a ***try-catch*** clause that includes a handler for the exception class.

The Core Java specification also defines certain abort-deferred regions that defer the transfer of control action, in particular *Atomic* scopes and ***finally*** clauses. Note however that once an ATC-enabled region has been entered, all method calls are susceptible to ATC other than the abort-deferred regions mentioned above. Since the Core Java specification rules prevent a Core program from making direct method calls to the Baseline Java domain, the problem does not arise of an ATC occurring at any point within "legacy" Baseline Java code that was not designed to expect that eventuality.

The ATC construct can be used for several common idioms, such as preventing overrun by timing out a sequence of actions, and for mode change. Special rules apply to the handlers for `ScopedException` to ensure that nested ATC scopes can be created without the danger of an outer ATC triggering exception being caught by an inner ATC ***catch*** clause.

3.5 Time

The Core Java specification defines a `Time` class that includes methods to construct times in all granularities from nanoseconds through to days. These can be used to program periodic timer events to trigger cyclic tasks or to timeout overrunning task execution.

In addition, the relative delay `sleep()` method and the absolute delay `sleepUntil()` method provide a programmatic means of coding periodic activity. In both cases, the time quantum can be specified to the nanosecond level.

There is also a method `tickDuration()` to return the length of a clock tick.

3.6 Other Features

The Core Java specification also includes a comprehensive low-level I/O interface for access to I/O ports, and a class `Unsigned` for unsigned relational operations. (Note that Baseline Java integral types are signed with regard to relational operations, but unsigned with regard to arithmetic.)

3.7 Profiles

A number of profiles of the Core specification are under development. One of the most interesting to the Ada community is the High Integrity Profile [Dobbing00] which 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 are very similar to those that steered the definition of the Ravenscar Profile [Dobbing98] and hence it is not surprizing that the high integrity profile provides similar functionality:

The dynamic Core Java execution environment is not supported (i.e. no direct interaction with a JVM);

All tasks are constructed during program startup;

`sleepUntil()` is supported, but `sleep()` is not;

Periodic, sporadic and interrupt event handler tasks are supported;

Signaling semaphores are supported, but counting semaphores and mutexes are not;

Protected objects are supported;

All synchronized code is locked using priority ceiling emulation (i.e. no priority inheritance)

The scheduling policy is FIFO_Within_Priorities;

Asynchronous abort is not supported;

Asynchronous dynamic priority change is not supported;

Asynchronous suspension is not supported;

Asynchronous transfer of control is not supported.

This profile allows the construction of a very small, fast, deterministic runtime system that could ultimately be a candidate even for formal certification.

A sub-group of the High Integrity Profile working group has been set up to examine how to apply and extend the profile to meet the needs of automotive control systems. This work is underpinned by the AJACS project ("Applying Java to Automotive Control Systems"). This is a two-year project partially funded by the European Commission to specify, develop and demonstrate an open technology that implements Java in deeply embedded interconnected Electronic Control Units in automotive applications such as engine control systems (see http://www.ajacs.org). The implementation of this technology is key to ensuring that the goals of the High Integrity Profile can be realized in practice."

4 RT JAVA EXPERT GROUP SPECIFICATION

4.1 Summary

Before setting out on the design of the real-time specification for Java ("RTSJ"), the RTJEG established the following guiding principles:

Applicability to particular Java environments. Usage is not to be restricted to particular versions of the Java Software Development Kit.

Backward compatibility. Existing Java code can run on any implementation of the RTSJ.

"Write Once, Run Anywhere". This is an important goal but difficult to achieve for real-time systems (as a trivial example of the difficulties, the correctness of a real-time program depends on the timing properties of the executing code, but different hardware platforms have different performance characteristics).

Current practice versus advanced features. The RTSJ attempts to address current real-time practice but also includes extensibility hooks to allow exploitation of new technologies.

Predictable execution. This is the highest priority goal; performance or throughput may need to be compromised in order to achieve it.

No syntactic extension. The RTSJ does not define new keywords or new syntactic forms.

Allow variation in implementation decisions. The RTSJ recognizes that different implementations will make different decisions (for example a tradeoff between performance and simplicity) and thus does not require specific algorithms.

The resulting specification consists of the `javax.realtime` package, an API whose implementation requires specialized support in the JVM. In summary, the design provides real-time functionality in several areas:

Thread scheduling and dispatching. The RTSJ introduces the concept of a *real-time thread* and defines both a traditional priority-based dispatching mechanism and an extensible framework for implementation-defined (and also user-defined) scheduling policies.

Memory management. The RTSJ provides a general concept of a *memory area* that may be used either explicitly or implicitly for object allocations. Examples of memory areas are the (garbage-collected) heap, and also "immortal" memory whose objects persist for the duration of an application's execution. Another important special case is a memory area that is used for object allocations during the execution of a dynamically determined "scope", and which is automatically emptied at the end of the scope. The RTSJ defines the concept of a "no-heap real-time thread" which is not allowed to reference the heap; this restriction means that such a thread can safely preempt the Garbage Collector.

Synchronization and resource sharing. The RTSJ requires the implementation to supply one or more mechanisms to avoid unbounded priority inversion, and it defines two monitor control policies to meet this requirement: priority inheritance and priority ceiling emulation. The specification also defines several "wait free queues" to allow a no-heap real-time thread and a Baseline Java thread to safely synchronize on shared objects.

Asynchrony. The RTSJ defines a general event model based on the framework found in the AWT and

Java Beans. An event can be generated from software or from an interrupt handler. Event handlers behave like threads and are schedulable entities. The design is intended to be scalable to very large numbers of events and event handlers (tens of thousands), although only a small number of handlers are expected to be active simultaneously. The RTSJ also defines a mechanism for asynchronous transfer of control ("ATC"), supporting common idioms such as timeout and mode change. The affected code needs to explicitly permit ATC; thus code that is not written to be asynchronously interruptible will work correctly.

Physical and "raw" memory access. The RTSJ provides mechanisms for specialized and low-level memory access. Physical memory is a memory area with special hardware characteristics (for example flash memory) and can contain arbitrary objects. Raw memory allows "peek" and "poke" of integral and floating-point variables at offsets from a given base address.

4.2 Concurrency and Synchronization

The basis of the RTSJ's approach to concurrency is the class `RealtimeThread`, a subclass of `Thread`.

4.2.1 Scheduling and Priorities

The RTSJ requires a base scheduler that is fixed-priority preemptive with at least 28 distinct priority levels, above the 10 Baseline Java levels. An implementation must map the 28 real-time priorities to distinct values, but the 10 non-real-time levels are not necessarily distinct.

Constructors for the `RealtimeThread` class allow the programmer to supply scheduling parameters, release parameters, memory parameters, a memory area, and processing group parameters. The scheduling parameters characterize the thread's execution eligibility (for example, its priority). A real-time thread can have a priority in either the real-time range or the Baseline Java range.

The release parameters identify the real-time thread's execution requirements and properties (whether it is periodic, aperiodic or sporadic). Memory parameters identify the maximum memory consumption allowed and an upper bound on the heap allocation rate (used for Garbage Collector pacing). Processing group parameters allow modeling a collection of aperiodic threads with bounded response time requirements.

Several release parameters classes are provided, corresponding to the kinds of real-time threads that are supported. A *periodic parameters* object specifies the period, the cost (maximum computation time per period), the deadline (which may be before the end of the period), and handlers for cost overrun and missed deadline. To create a periodic thread, the programmer constructs a real-time thread with a periodic parameters object as its release parameters. The `run()` method for such a thread should invoke the method `waitForNextPeriod()` to suspend itself after its per-period work. The programmer may supply overrun handlers to respond to two kinds of abnormality: overrunning the budgeted cost (detecting this situation requires support from the underlying platform and is therefore not required of the implementation unless such support exists), and missing a deadline. The overrun or miss handlers can invoke the `schedulePeriodic()` method to resume scheduling of the periodic thread.

An *aperiodic parameters* object is used for schedulable entities that may become active based on the occurrence of an asynchronous event or the invocation of a `notify()` or `notifyAll()`. The aperiodic parameters define the cost, deadline, overrun handler, and miss handler analogously to periodic parameters.

A special case of aperiodic parameters is *sporadic parameters*. A schedulable entity constructed with sporadic parameters has a minimum inter-arrival time between releases.

One of the RTSJ's distinguishing capabilities is a general-purpose extensible scheduling framework. An instance of the `Scheduler` class manages the execution of schedulable entities and may implement a feasibility analysis algorithm. Through method calls a real-time thread can be added to, or removed from, the scheduler's feasibility analysis; the release parameters are used in this analysis. The scheduler's `isFeasible()` method returns `true` if the existing schedulable entities are schedulable (i.e., will always meet their deadlines) and `false` otherwise.

The priority-based scheduler is required to be the default scheduler at system startup, but the programmer can modify this at run time (for example setting an Earliest-Deadline First scheduler, if one is supplied by the implementation).

The priority-based scheduler is FIFO within priorities and manages Baseline Java threads as well as real-time threads. When a thread blocks it goes to the tail of the blocked queue for its priority level, for the resource on which it is blocked. When a blocked thread becomes ready to run, or when a running thread invokes `yield()`, it goes to the tail of the ready queue for its priority level. When a thread's priority is modified explicitly through a method call the thread goes to the tail of the relevant queue for its new priority level. When a thread is preempted, the RTSJ does not specify where in the ready queue for its priority the thread is placed. This nondeterminism does not affect feasibility analysis.

The priority-based scheduler is said to be fixed-priority since it is not allowed to modify thread priorities

implicitly except for priority inversion avoidance (see below). Thus schemes such as "priority aging" are not allowed. Time slicing of the highest-priority threads is permitted, although the implementation provides no explicit support for such a policy.

The general scheduling framework was motivated by several considerations. First and foremost was the user requirement: the RTSJ is intended for a large variety of applications, and limiting the scheduling policies would have been too constraining. Second, the flexibility fits in well with Java's dynamic model. Third, several members of the RTJEG, in particular the original spec lead Greg Bollella (then at IBM), were experts in this area on both the theoretical and practical sides.

4.2.2 Synchronization

An unbounded priority inversion in a thread synchronizing on a locked object can lead to missed deadlines, and the RTSJ accordingly requires that the implementation supply one or more monitor control policies to avoid this problem. By default the policy is priority inheritance, but the RTSJ also defines a priority ceiling emulation policy. Each policy can be selected either globally or per-object and the choice can be modified at run time. An implementation can supply a specialized form of priority ceiling emulation that prohibits a thread from blocking while holding a lock; this avoids the need for mutexes and queues in the implementation.

A subtle problem seems to arise if a Baseline Java thread and a no-heap real-time thread attempt to communicate through a synchronized object (such an object cannot be in the heap, but it may be in immortal memory). The apparently troublesome scenario is the following:

1. The regular thread locks the object.
2. The garbage collector preempts and starts to run.
3. The no-heap real-time thread preempts the garbage collector, with the heap possibly in an inconsistent state.
4. The no-heap real-time thread attempts to synchronize on the object currently locked by the regular thread.
5. The regular thread inherits the no-heap real-time thread's priority and resumes execution.
6. The regular thread allocates an object in the heap, but this can corrupt the heap (which might be in an inconsistent state).

In fact the RTSJ semantics prevent this problem although at the price of extra latency for the no-heap real-time thread. When the regular thread attempts to allocate an object at step 6, it will not be able to do so since the garbage collector holds the lock on the heap. Thus the garbage collector will have its priority boosted (by priority inheritance), and when it releases the lock the heap will be in a consistent state so that the Baseline Java thread can continue in its "critical section" of code synchronized on the object it is sharing with the no-heap real-time thread.

The price for a consistent heap is extra latency, since the no-heap real-time thread now needs to wait for the Garbage Collector. The RTSJ allows the programmer to avoid this latency through wait-free queues; Baseline threads and no-heap real-time threads can use such queues to communicate without blocking.

4.3 Memory Management

Perhaps the most difficult issue for the RTSJ was the question of how to cope with garbage collection ("GC"). Requiring specific GC performance or placing constraints on GC-induced thread latency would have violated several guiding principles. Instead the opposite approach was taken: the RTSJ makes no assumptions about the GC algorithm; indeed in some environments there might not even be a garbage collector.

The key concept is the notion of a *memory area*, a region in which objects are allocated. The garbage-collected heap is an example of a memory area. Another memory area is *immortal memory*: a region in which objects are not garbage collected or relocated and thus persist for the duration of the program's execution. More flexibility is obtained through *scoped memory areas*, which can be explicitly constructed by the programmer. Each scoped memory area contains objects that exist only for a fixed duration of program execution. The heap and immortal memory can be used by either regular threads or real-time threads; scoped memory can be used only by real-time threads.

Common to any memory area is an `enter()` method which takes a `Runnable` as a parameter. When `enter()` is invoked for a memory area, that area becomes active, and the `Runnable` object's `run()` method is invoked synchronously. The memory area is then used for all object allocations through "`new`" (including those in methods invoked from `run()` whether directly or indirectly) until either another memory area becomes active or the `enter()` method returns. When `enter()` returns, the previous active area again becomes active.

A memory area may be provided to a real-time thread constructor; it is then made active for that real-time thread's `run()` method when the thread is started.

Memory areas may also be used for "one shot" allocation, through factory methods that construct objects or arrays in the associated area.

Scoped memory may be viewed as a generalization of a method's stack frame. Indeed, early in the design the RTJEG considered providing a mechanism through which objects allocated in a method would be stored on the stack instead of the heap, with automatic reclamation at method exit instead of garbage collection. Standard class libraries could then be rewritten with the same external specifications (public members, method signatures and return type) but with an implementation that used the stack versus the heap for objects used only locally. To prevent dangling references a check would be needed (no assignment of a stack object reference where the target reference is longer lived than the source). Some sort of check (either at compile time or run time) is inevitable. However, the reason that a simple stack-based object scheme was eventually rejected is that a reference to a local object could not be safely returned to a caller. Thus the goal of using specially-implemented versions of existing APIs would not be achievable.

Instead the RTSJ has generalized the concept of storing local objects on the stack. A scoped memory area is used not just for one method invocation but for the "closure" of all methods invoked from a `Runnable`'s `run()` method. The objects within the memory area are not subject to relocation or collection, and an assignment of a scoped reference to another reference is checked (in general at run time) to prevent dangling references. Scopes may be nested: while one scoped memory area is active, another may be entered, and in fact the same scoped memory area may be entered while in use by an outer scope. When the outermost scope is exited (i.e., when the earliest `enter()` for a given scoped memory area returns) the area is reset so that it contains no objects. A common idiom is a `while` or `for` loop that invokes `enter()` on a scoped memory area at each iteration. All objects allocated during the iteration are effectively flushed when `enter()` returns, so there is no storage leakage. The entire memory area is released when it is no longer accessible. In general the implementation needs to use a reference count scheme or its equivalent for this purpose.

The RTSJ provides two main non-abstract classes for scoped memory: "LT" memory (linear time) and "VT" memory ("variable time"). Object allocation and default initialization for LT memory must be implemented to be linear in the size of the object; no such constraint is imposed on VT memory. In practice, the difference between the two is that the implementation must allocate the entire memory region used for LT memory (although not necessarily contiguously) whereas for VT memory only an initial region needs to be allocated in advance, with further chunks added as necessary.

The RTSJ also provides several more specialized kinds of memory area. Support for physical memory (i.e. memory with special characteristics) is offered through *immortal physical memory* and *scoped physical memory*. This can be useful for efficiency; for example the programmer may want to allocate a set of objects in a fast-access cache. The *raw memory access* and *raw memory float access* memory areas offer low-level access ("peek" and "poke") to integral and floating-point data, respectively.

4.4 Asynchrony

The RTSJ supplies two mechanisms relevant to asynchronous communication: asynchronous event handling, and asynchronous transfer of control.

4.4.1 Asynchronous Event Handling

The RTSJ defines the concepts of an *asynchronous event* and an *asynchronous event handler*, and it specifies the relationship between the two.

An async event can be triggered either by a software thread or by a "happening" external to the JVM. The programmer can associate any number of async event handlers with an async event, and the same handler can be associated with any number of events. Async event handlers are schedulable entities and are constructed with the same set of parameters as a real-time thread; thus they can participate in feasibility analysis, etc. However, there is not necessarily a distinct thread associated with each handler. The programmer can use a bound async event handler if it is necessary to dedicate a unique thread to a handler.

When an async event is fired, all associated handlers are scheduled. A programmer-overridable method on the handler establishes the behavior. If the same event is fired multiple times, the handler's actions are sequentialized. In the interest of efficiency and simplicity, no data are passed automatically from the event to the handler. The programmer can define the logic necessary to buffer data, or to deal with overload situations where not all events need to be processed.

The async event model uses the same framework as event listeners in Java Beans and the AWT but generalizes and formalizes the handler semantics with thread-like behavior.

4.4.2 Asynchronous Transfer of Control

Asynchronous Transfer of Control ("ATC") is a mechanism whereby a triggering thread (possibly an async event handler) can cause a target thread to branch unconditionally, without any explicit action from the target thread. It is a controversial capability. The triggering thread does not know what state the target thread is in when the ATC is initiated while, on the other side, the target thread needs to be coded very carefully if it is susceptible to ATC. ATC also imposes a run-time

cost even for programs that do not use the functionality. Nevertheless, there are situations in real-time programs where the alternative style (polling for a condition that can be asynchronously set) induces unwanted latency, and the user community identified several situations (timing out on an operation, or mode change) where ATC offers the appropriate semantic framework.

A rudimentary ATC mechanism was present in the initial version of the Java language: the `Thread` methods `stop()`, `destroy()`, `suspend()` and `resume()`. Unfortunately a conflict between the ATC semantics and program reliability led to these methods' deprecation (`stop()`, `suspend()`, `resume()`) or stylistic discouragement (`destroy()`). If a thread is `stop`ped while it holds a lock, the synchronized code is exited and the lock is released, but the object may be in an inconsistent state. If a thread is `destroy`ed while it holds a lock, the lock is not released, but then other threads attempting to acquire the lock will be deadlocked. If a thread is `suspend`ed while it holds a lock, and the resuming thread needs that lock, then again a deadlock will ensue.

The problem is that Baseline Java does not have the Ada concept of an "abort-deferred region". The RTSJ has introduced this concept, together with other semantic constraints, in the interest of providing ATC that is safe to use.

Several guiding principles underlie the ATC design:

Susceptibility to ATC must be explicit in the affected code.

Even if code allows ATC, in some sections ATC must be deferred — in particular, in synchronized code.

An ATC does not return to the point where it was triggered (i.e. it is a "goto" rather than a subroutine call), since with resumptive semantics an arbitrary action could occur at arbitrary points.

If ATC is modeled through exception handling, the design needs to ensure that the exception is not caught by an unintended handler (for example a method with a catch clause for `Throwable`)

ATC needs to be expressive enough to capture several common idioms, including time-out, nested time-out (with correct disposition when an "outer" timer expires before an "inner" timer),mode change, and thread termination.

From the viewpoint of the target thread, ATC is modeled by exception handling. The class `AsynchronouslyInterruptedException` (abbreviated "AIE") extends `InterruptedException` from `java.lang`. An ATC is initiated in the target thread by a triggering thread causing an instance of AIE to be thrown. This is not done directly, since there is no guarantee that the target thread is executing in code prepared to catch the exception. In any event there is no syntax in Java for one thread to asynchronously throw an exception in another thread[1].

ATC only occurs in code that explicitly permits it. The permission is the presence of a "throws AIE" clause on a method or constructor. ATC is deferred in methods or constructors lacking such a clause, and is also deferred in synchronized code.

The basic ATC construct is the `doInterruptible()` method of AIE. This method takes an `Interruptible` as parameter; the `Interruptible` interface defines the abstract methods `run()` (which has a "throws AIE" clause) and `interruptAction()`. The target thread constructs an AIE instance *aie*, makes this instance available to a triggering thread, and then invokes *aie*.`doInterruptible(`*obj*`)` on an `Interruptible` object *obj*; this causes *obj*.`run()` to be invoked synchronously. If the triggering thread invokes *aie*.`fire()` while the target thread is still executing `run()`, the target thread will be asynchronously interrupted as soon as it is outside of ATC-deferred code, `run()` will return, and the target thread will invoke *obj*.`interruptAction()`. Note that the throwing and handling of the AIE are encapsulated in the implementation of the `fire` and `doInterruptible` method. Calling `fire()` too early (before `doInterruptible` has been invoked) or too late (after `run` has returned) has no effect on the target thread.

The `Timed` class (a subclass of AIE) is provided as a convenience to deal with time out; the firing of the AIE is done by an implementation-provided async event handler rather than an explicit user thread.

The RTSJ's analog of `Thread.stop` is for a triggering thread to invoke `interrupt()` on a real-time thread that is to be terminated. The effect of `interrupt()` on a real-time thread is a generalization of the effect on a regular thread. If `interrupt()` is invoked on a regular thread, an `Interrupted-Exception` will be thrown when the thread is blocked. If `interrupt()` is invoked on a real-time thread, an AIE will be thrown when the thread is in asynchronously interruptible code. (Deferring the interruption in synchronized code avoids the problem that led to the deprecation of `Thread.stop`.) Moreover, since the AIE remains pending even if the exception is caught (unless logic in the handler explicitly disables the propagation) the effect of invoking

[1] The functionality is actually present in `Thread.stop()`, but this method is now deprecated.

`interrupt()` on a real-time thread will be to terminate the thread; the latency depends on the duration of non-ATC code in the method call stack.

4.5 Time and Timers

The RTSJ provides several ways to specify high-resolution (nanosecond accuracy) time: as an *absolute* time, as a *relative* number of milliseconds and nanoseconds, and as a *rational* time (a frequency, i.e. a number of occurrences of an event per relative time). In a relative time 64 bits (a `long`) are used for the milliseconds, and 32 bits (an `int`) for the nanoseconds.

The rational time class is designed to simplify application logic where a periodic thread needs to run at a given frequency. The implementation, and not the programmer, needs to account for round-off error in computing the interval between release points.

The time classes provide relevant constructors, arithmetic and comparison methods, and utility operations. These classes are used in constructors for the various release parameters classes.

The RTSJ defines a default real-time clock which can be queried (for example to obtain the current time) and which is the basis for two kinds of timers: a one-shot timer, and a periodic timer. Timer objects are instances of async events; the programmer can register an async event handler with a timer to obtain the desired behavior when the event is fired. A handler for a periodic timer is similar to a real-time thread with periodic release parameters but is likely to be more efficient.

4.6 Other Features

The RTSJ provides a real-time system class analogous to `java.lang.System`, with "getter" and "setter" methods to access the real-time security manager and the maximum number of concurrent locks. It also supplies a binding to Posix signal handlers (required of the implementation if the underlying system supports Posix signals).

4.7 Status

The initial version of the RTSJ was published in June 2000 after a 15-month design. A Reference Implementation was initiated by Timesys in early 2001, and experience from that effort has resulted in some suggested modifications to the specification. Work on a final version of the specification, with accompanying Reference Implementation and Compatibility Test Suite, are currently in progress.

5 COMPARATIVE ANALYSIS

5.1 The Two RT Java Specifications

The main distinction between the two specifications is in their execution environment models.

The Core Java specification approach is to build a Core program as a distinct entity from a Java virtual machine. The intent is for the Core Java specification to be used to build small, fast, high performance stand-alone programs that have been traditionally written in C, C++ and Ada. These programs may communicate with a virtual machine in a controlled way.

The RTSJ approach is to define an API with real-time functionality that can be implemented by a specially constructed Java virtual machine. The intent is for the RTSJ specification to be used to build predictable real-time threads that execute in the same environment as non-real-time threads within one virtual machine.

It is interesting to conclude that a system could be composed of sub-systems that are implemented using both specifications. For example, a system may require a high-performance micro kernel implemented using the Core Java specification, executing in conjunction with a JVM that is executing some predictable real-time threads, as well as using a wide range of standard APIs within background threads.

This distinction in the execution environment model is also reflected in the goals and semantics of the specifications, for example:

The RTSJ specification is more of a scalable framework that can be implemented by a wide variety of virtual machines with differing characteristics, and executing over a variety of operating systems. In contrast, the Core specification has more precise and fixed semantics that match the characteristics of traditional real-time kernels.

The RTSJ specification retains security of operation, for example by preventing dangling references, and by ensuring that ATC is deferred in synchronized code. This is consistent with Java design philosophy and the safety model of JVMs. In contrast, the Core specification assumes that the Core programmer is a "trusted expert" and so provides more freedom and less safety; for example a dangling reference to an object in a released allocation context can occur; an ATC can trigger immediately within a priority-ceiling-locked protected object; and the `stop()` method does not unlock mutexes or release semaphores.

The RTSJ specification concentrates on adding predictability to JVM thread operations, but does not aim to deal with memory footprint, performance, or interrupt

latency. In contrast, the Core specification has been designed to optimize on performance, footprint and latency. Kelvin Nilsen has summarized this distinction as follows: *"The RTSJ makes the Java platform more real-time, whereas the Core Java specification makes real-time more Java-like."*

The other major distinction between the two specifications is in their licensing models. The RTSJ specification is an extension to the trademarked Java definition and hence is subject to Sun Microsystems, Inc licensing requirements. However the Core specification is independent of the trademark (and hence licensing requirements) and is being put forward as an ISO standard specification via the J Consortium's approval to be a submitter of ISO Publicly Available Specifications.

5.2 Comparison with Ada95

5.2.1 Similarity to Ada Real-Time Annex

Almost all new elements in the two real-time Java specifications can be found in either the Ada95 core language definition, or its Systems Programming or Real-Time Annex [Ada95]. These include:

A guaranteed large range of priority values;

Well-defined thread scheduling that must include FIFO_Within_Priorities policy;

Addition of Protected Objects to the existing Synchronized objects and methods, that (may) prohibit voluntary suspension operations, and that define a Ceiling Priority for implementation of mutual exclusion (c.f. `Ceiling_Locking` policy);

Addition of asynchronous transfer of control triggered by either time expiry or an asynchronous event;

Allocation of, and access to, 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 (c.f. `Ada.Dynamic_Priorities`);

Absolute time delay (c.f. *delay_until* statement);

Use of nanosecond precision in timing operations (c.f. `Ada.Real_Time.Time`);

Definition of interrupt handlers and operations for static and dynamic attachment.

In addition, the High Integrity Profile of the Core Java specification has the same execution model as that of the Ravenscar Profile, as discussed in section 3.7.

Thus the real time extensions for Java are quite compatible with the Ada95 Real-Time Annex and Ravenscar Profile execution models, which encourages the view that both Ada and Real-Time Java implementations could be used to develop parallel subsystems that execute in a common underlying environment.

5.2.2 Dissimilarity to Ada R-T Annex

The following design decisions were taken during the development of the Core Java specification 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 Core Java specification, 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 environment to support both priority 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, thereby providing a single mutual exclusion mechanism that is optimal for static timing analysis.

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 operation. 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 for all protected object 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 Core Java specification. 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 transfer of control) which could be useful for example to ignore an execution time overrun signal in certain context-specific situations. This option is not provided by Ada95.

Dangling references to objects within allocation contexts can occur in the Core Java specification. Ada95 semantics were carefully crafted to prevent dangling references except via unchecked programming.

Some of the RTSJ design decisions that conflict with the Ada 95 core language and Real-Time Annex follow:

The RTSJ has a more general view of scheduling and dispatching, with feasibility analysis, overrun and deadline miss handlers, rational time, etc.

For the fixed-priority preemptive policy, the RTSJ does not dictate where in the ready queue a preempted thread is placed. In the Ada Real-Time Annex, this is deterministic (the preempted task is placed at the head of the queue for its priority).

The RTSJ's priority ceiling emulation monitor control policy requires queuing in one supported model that allows a thread holding a priority ceiling lock to block.

There is no direct Ada analog to the RTSJ's async event model (in particular the many-to-many relationship between events and handlers).

In the RTSJ, an ATC is not deferred in finally clauses (this is because the bytecodes do not directly reflect where finally clauses were present in the source code). In Ada, abort is deferred during finalization.

6 LOOKING AHEAD

We can look back on the '90s as the decade of revolutionary communication for individuals and for business, primarily via the internet. Use of e-mail, the web, mobile phones, e-banking etc has become part of everyday life, and e-business is an extremely rapidly growing industry. The Java execution environment has been most prominent in the software part of this new technology, with its write-once-run-anywhere capability inherent in its bytecodes and in the JVM, and with its abundant highly practical and portable APIs. But many of today's Java applications do not have demanding size and performance constraints.

So what will the next decade bring us? The next revolution could well be in communicating embedded devices. Some have predicted a trillion communicating devices by 2025, affecting almost all aspects of our daily lives. In at least some of these cases, an embedded device application environment will have demanding size and performance constraints, and will also require high availability, high integrity and hard real time deadlines. A growing number of these systems may even have safety critical requirements.

The Real-Time Java initiatives presented in this paper illustrate that the Java community as a whole, and its tool vendors and those who promote international standards in particular, are taking real-time requirements and embedded system constraints very seriously, and are preparing Java, its JVMs and its APIs for the next revolution. So what of Ada95, or its next revision Ada0Y?

Ada enthusiasts can argue quite validly that Ada95 environments can already meet the stringent requirements of embedded systems better than any other, and that Ada's suitability for use in high integrity and safety critical is second to none. However it is clear that Ada did not figure in the communications revolution of the 90's, and does not enter the new millennium with an expanding community. So can Ada, with all its excellent reputation within high integrity and safety critical embedded systems, find a role in the new revolution as the battleground moves into Ada's own strongholds?

The key to Ada's successful future almost certainly lies in seamless co-operation with the Java environment, rather than in competing with it. It is interesting to see how the two language environments are starting to converge somewhat. This cross-fertilization could be known as the "*Jada effect*".

We have already seen in this paper that many of the new ideas for Real-Time Java have been borrowed from Ada, such as those needed for predictability and deterministic schedulability analysis. Baseline Java had already used Ada's exception model, and now we see that the real-time extensions have equivalents for protected objects including entries, priority ceiling emulation, well-defined thread scheduling policies, absolute delay, high precision timers, suspension objects, dynamic priority change, interrupt handlers, asynchronous transfer of control, access to physical memory, abort-deferred regions etc. So Java is definitely evolving towards Ada in the real-time domain.

In similar fashion, Ada is evolving towards Java. The Ada95 revision already brought in support for a comprehensive object oriented programming model not

dissimilar to that in Java, including single inheritance hierarchies, constructors and finalizers etc. The next revision of Ada (Ada0Y) may well see the addition of support for Java-style *interfaces*, thereby providing the same limited form of multiple inheritance as in Java (from one class plus any number of interfaces). Furthermore, Ada0Y may relax the rules that currently prevent mutually-dependent package specifications, via a new *with type* construct. This would allow mutually-dependent Java classes to be modeled as Ada packages that each define a tagged type plus its primitive operations, without having kludges to workaround circularities in the "with" dependencies. Finally there is even some discussion about whether to allow a Java-like OOP syntax for invoking the primitive operations of a tagged type. This could be used instead of the traditional procedure calling style that requires some rules to identify which parameter is the object that controls the dynamic dispatching, replacing it with an OOP style along the lines of *object'Operation(parameters)*.

So it seems that both Java and Ada are undergoing the *Jada* effect. However as well as language convergence, it is also very important to have execution environment convergence if the two are going to co-exist happily. We have already seen some worthy attempts at integration between Java and Ada execution environments. A few different approaches are mentioned below:

Aonix's *AdaJNI* [Flint00] makes use of the Java Native Interface that is provided with the Java Development Kit. This approach allows Ada native programs to interact with Java classes and APIs that are executed by a local or remote JVM via Ada-style interface packages.

Ada Core Technologies *JGNAT* [ACT00] compiles Ada95 into Java bytecodes in standard class files. This approach allows JVM-based programs to comprise a mixture of Ada and Java code. Again, there is also the capability for the Ada code to access Java classes and APIs via Ada-style interface packages.

Ada ORB vendors e.g. [OIS] provide access to CORBA objects from Ada programs. This approach allows a logically distributed, mixed-language (including Ada and Java) system to communicate using the CORBA client/server model.

If Ada is to gain any kind of foothold in the new generation of communicating devices, we must build on foundations such as these. The efforts of users and vendors alike within the Ada community need to be focused on developing and evolving Ada in ways that are compatible with the emerging requirements, not least a seamless co-existence with the new Real-Time Java execution environments, their JVMs and their APIs. If we can achieve this goal, this can give a whole new lease of life to Lady Ada. We may even want to rename her Lady Jada ☺.

7 ACKNOWLEDGEMENTS

The authors appreciate the helpful comments of two anonymous referees.

8 REFERENCES

[ACT00] Ada Core Technologies, Inc; *JGNAT User's Guide*; 2000.

[Ada95] *Ada95 Reference Manual*, International Standard ANSI/ISO/IEC-8652:1995, Jan 1995.

[Bollella00] Bollella G., Gosling J., Brosgol B., Dibble P., Furr S., Hardin D., and Turnbull M.; *The Real-Time Specification for Java*; Addison-Wesley; 2000.

[Brosgol98] Brosgol B.; *A Comparison of the Concurrency and Real-Time Features of Ada and Java*; Proceedings of Ada U.K. Conference 1998; Bristol, U.K.

[Dobbing98] Dobbing B. and Burns A., *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).

[Dobbing00] Dobbing B. and Nilsen K., *Real-Time and High Integrity Extensions to Java™*, Embedded Systems Conference West 2000 Proceedings, September 2000.

[Flint00] Flint S. and Dobbing B., *Using Java™ APIs with Native Ada Compilers,* In "Reliable Software Technologies - Ada-Europe 2000", Lecture Notes in Computer Science 1845, Springer Verlag (June 2000).

[JCons00] International J Consortium Specification, *Real-Time Core Extensions*, Draft 1.0.14, September 2nd 2000. Available at http://www.j-consortium.org.

[JLS00] Gosling J., Joy B., Steele G., and Bracha G.; *The Java Language Specification* (second edition); Addison-Wesley; 2000.

[Jones97] Jones R. and Lins R.; *Garbage Collection*; Wiley and Sons; 1997.

[Nilsen96] Nilsen K.; *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.

[NIST99] Nilsen K., Carnahan L., and Ruark M., 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.

[OIS] Objective Interface Systems, Inc, *ORBexpress*, http://www.ois.com