

# Is Java Augmented with the RTSJ a Better Real-Time Systems Implementation Technology than Ada 95?

Andy Wellings  
Department of Computer Science, University of York  
Heslington, York YO10 5DD, UK  
andy@cs.york.ac.uk

August 22, 2003

## Abstract

This paper argues that Java (augmented by version 1.1 of the Real-Time Specification) will become superior to Ada 95 as a real-time systems implementation technology – once efficient and full implementations of version 1.1 become available. This is because the RTSJ will have an integrated set of mechanisms for representing the characteristics of real-time activities and a framework from within which those activities can be scheduled. Ada 95, in contrast, has a set of orthogonal low-level mechanisms which are invisible to the scheduler. Ada must, therefore, rise to the challenge and enhance its real-time support.

## 1 Introduction

Since its inception, Ada has been targeted at real-time systems development. The current version of the language has a mature approach to handling priority-based systems. Implementations are stable and the Ravenscar Profile [2] allows the production of efficient and predictable programs and their supporting run-time environments. Of the current real-time technology, Ada is technically superior to its competitors. However, Ada must not rest on its laurels. The Real-Time Specification for Java [1] continues to gather momentum, and the next major release of the specification promises to be better defined and more integrated<sup>1</sup>. Implementations are beginning to emerge (for example, [5]) and, in time, they will mature into efficient products that support the full revised specification. A Ravenscar-Java profile has been proposed [4] and a Java Expert Group

---

<sup>1</sup>It is anticipated that Version 1.0.1 of the Specification will appear later this year. This version will remove most of the ambiguities in the current specifications and correct known errors. Version 1.1 of the Specification will address major shortcomings and is likely to be produced in the latter part of 2004. The term next revision is used in this paper to mean the 1.1 Version.

is currently being formed to consider the use of Java, the RTSJ and Ravenscar Java for high integrity systems. Indeed, Java with the RTSJ is beginning to threaten the only remaining strongholds of Ada.

The aim of this paper is to be provocative, and help stimulate discussion at the workshop on future Ada support. The goal is to challenge complacency and to motivate change in Ada, rather than to defend Ada's well-known strengths. Consequently, the paper makes the following proposition:

*Java augmented by the next version of the RTSJ will have better support for real-time systems development than Ada because it will have an integrated set of mechanisms for representing the characteristics of real-time activities and a framework from within which those activities can be scheduled. Ada 95, in contrast, has a set of orthogonal low-level mechanisms.*

Hence, the paper contends that Java with the next version of the RTSJ will have more direct expressive power in its support for real-time than Ada 95 does. The presence of ambiguity (and lack of clarity) in the current specification, the known limitations on some of the functionality provided and the absence of efficient implementations of the full specification are currently inhibiting the wide-spread take-up of Java and the RTSJ. However, it is only a matter of time before these problems are rectified. Ada, therefore, *must* continue to enhance its real-time facilities if it is to retain its technical superiority.

The paper start by discussing scheduling models, and identifies the key characteristics that are needed for real-time scheduling. Three suppositions are made which will be used to defend the above proposition. Sections 3 and 4, then summarise the facilities provided by the RTSJ and Ada respectively. Finally, section 5 concludes by comparing the two approaches and using the suppositions made in Section 2, defends the proposition.

## 2 Scheduling Models for Real-Time Systems

Scheduling consists of three components [3]

- an algorithm for ordering access to resources (scheduling policy)
- an algorithm for allocating the resources (scheduling mechanism)
- a means of predicting the worst-case behaviour of the system when the policy and mechanism are applied (schedulability analysis - called feasibility analysis by the RTSJ).

Once the worst-case behaviour of the system has been predicted, it can be compared with the system's timing requirements to ensure that all deadlines will be met.

There have been many different scheduling approaches developed over the last 10-15 years [3], for example, cyclic scheduling, fixed priority scheduling, earliest deadline first, value-based, etc. Most current real-time programming languages and operating systems in widespread use today support fixed priority scheduling.

Modern approaches to scheduling view the system as consisting of a number of schedulable objects (often called threads, processes, tasks, event handlers). Each schedulable object is characterized by the following [6]:

**Release profile.** Typically after a schedulable object is started, it waits to be released (or may be released immediately); when released it performs some computation and then waits to be released again (the time at which it waits is often called its completion time). The release profile defines the frequency with which the releases occur; they may be time triggered or event triggered. When time triggered releases occur on a regular basis, they are called periodic releases. Event-triggered releases are typically classified into sporadic (meaning that they are irregular but with a minimum inter-arrival time) or aperiodic (meaning that no minimum inter-arrival assumptions can be made, although other information may be available such as the distribution of the releases).<sup>2</sup> Once a schedulable object has been released, it is eligible for execution. During its execution, it may be blocked waiting for a resource (for example, a mutual exclusion lock). When the resource becomes available, the schedulable object is again eligible for execution.

---

<sup>2</sup>Of course, time-triggered sporadic releases or event triggered periodic releases are also possible.

**Processing cost per release.** This is some measure of how much of the processor's time is required to execute the computation associated with the schedulable object's release (this may be a worst-case value or an average value depending on the feasibility analysis being used). It is often referred to as a CPU budget.

**Other hardware resources required per release.** This is some measure of the hardware resources needed (other than the processor). For networks, it is usually the time needed (or bandwidth required) to send the schedulable object's messages across the network. For memory, it is the amount of memory required by the schedulable object (and if appropriate, the types of memory).

**Software resources required per release.** This is a list of the non-shareable resources that are required for each release of the schedulable object and the processing cost of using each resource. Access to non-shareable resources is a critical factor when performing schedulability analysis. This is because non-shareable resources are usually non-preemptible. Consequently, when a schedulable object tries to acquire a resource, it may be blocked if that resource is already in use. This blocking time has to be taken into account in any analysis. If the list of software resources is not available then a maximum blocking time must be provided.

**Deadline.** The time that the schedulable object has to complete the computation associated with each release. Usually only a single deadline is given, therefore, the time is a relative value rather than an absolute value. Where the deadline of a schedulable object is greater than its minimum period between releases (or it has overrun its deadline, and the application has decided to let it continue), the schedulable object may be released even though the execution associated with the previous release has not completed. In this case, when the schedulable object does complete, it is immediately re-scheduled for execution (re-released).

**Value.** A metric that indicates the schedulable object's contribution to the overall functionality of the application. It may be a very coarse indication (such as safety critical, mission critical, non critical), a numeric value giving a measure for a successful meeting of a deadline, or a time-valued function which takes the time at which the schedulable object completes and returns a measure of the value (for those systems where there is no fixed deadline).

The information needed for the above characteristics comes from a variety of sources. It may be specified as part of the application's requirements (for example, the value parameter), derived during the system design (for example, the deadline parameter), or result from static or dynamic analysis of the final code (for example, the cost and blocking time parameters).

## 2.1 Suppositions

In order to support the proposition that Java augmented by the RTSJ has more expressive power than Ada and that it will become a superior implementation technology to Ada (once efficient implementations emerge), the following suppositions are made.

### Supposition 1

*A real-time systems implementation technology should allow the programmer to express the characteristics of schedulable objects directly in the program and in a way that can be made available to the scheduler.*

Without this, it is not possible for the scheduler to perform any on-line analysis or for it to be able to respond to dynamically changing environments.

### Supposition 2

*The scheduler must ensure that any run-time deviation from a schedulable object's characteristics does not undermine any feasibility analysis that has been performed (off-line or on-line).*

Without this, it is not possible for the scheduler to protect schedulable objects of high criticality from those with low criticality.

### Supposition 3

*The scheduler must provide mechanisms that can bound the impact of schedulable objects which have non-periodic release profiles.*

Without this, it is not possible for the scheduler to run aperiodic schedulable objects as anything other than background activities.

If Suppositions 1, 2 and 3 are accepted, there are requirements placed on a real-time systems implementation technology to allow the expression of a schedulable object's characteristics (presented above) and to integrate the supporting mechanisms

with the run-time scheduler. A technology that supports a wide range of these characteristics and integrates that support with the run-time scheduler is, therefore, superior to one that does not.

## 3 The RTSJ Approach

The RTSJ incorporates the notion of a schedulable object rather than simply considering threads. A schedulable object is any object that implements the `Schedulable` interface. The current specification essentially provides two types of object that implement this interface, `RealtimeThreads` and `AsyncEventHandlers`. Objects that implement the `Schedulable` interface have the following associated attributes (represented by classes).

**ReleaseParameters** - Giving the processing cost for each release (its CPU budget) of the schedulable object and its deadline; if the object is released periodically or sporadically then subclasses allow an interval to be given (and enforced for sporadics). Event handlers can be specified for the situation where a deadline is missed or where the processing resource consumed becomes greater than the cost specified. However, note that there is no requirement for a real-time JVM to monitor the processing time consumed by a schedulable object. If it does, then there is a requirement that a schedulable object be given no more than 'cost' processing units each release<sup>3</sup>. It should be noted that Version 1 of the RTSJ makes no mention of blocking time in any of the parameters associated with schedulable objects. The assumption is that a particular implementation will subclass `ReleaseParameters` to bring in this data. Typically, this will be a relative time value set by the programmer as a result of off-line analysis of the code. It is likely that Version 1.1 will formally introduce blocking time into the `ReleaseParameters` class.

**SchedulingParameters** - The `SchedulingParameters` class is empty; however subclasses allow the priority of the object to be specified along with its importance (value) to the overall functioning of the application. Although the RTSJ specifies a minimum range of real-time priorities (28), it makes no statement on the allowed values of the importance parameter.

**MemoryParameters** - Giving the maximum amount of memory used by the object in its default memory area, the maximum amount of memory used in immortal memory, and a maximum al-

---

<sup>3</sup>The current version of the specification has limited functionality for supporting this model across all schedulable objects. It practice only real-time threads with periodic release parameters are catered for. Version 1.1 of the specification is likely to provide additional features in this area.

location rate of heap memory. An implementation of the RTSJ is obligated to enforce these maximums and throw exceptions if they are violated.

**ProcessingGroupParameters** - This allows several schedulable objects to be treated as a group and to have an associated period, cost and deadline. When used with aperiodic activities, they allow their impact to be bounded.

The methods in the **Schedulable** interface can be divided into three groups.

- Methods that will communicate with the scheduler and will result in the scheduler either adding or removing the schedulable object from the list of objects it manages (called its feasibility set), or changing the parameters associated with the schedulable object (but only if the resulting system is feasible).
- Methods that get or set the parameter classes associated with the schedulable object. If the parameter object set is different from the one currently associated with the schedulable object, the previous value is lost and the new one will be used in any future feasibility analysis performed by the scheduler. Note, these methods do not result in feasibility analysis being performed and the parameters are changed even if the resulting system is not feasible.
- Methods that get or set the scheduler. For systems that support more than one scheduler, these methods allow the scheduler associated with the schedulable object to be manipulated

Changing the parameters of a schedulable object whilst it is executing can potentially undermine any feasibility analysis that has been performed, and cause deadlines to be missed. Consequently, the RTSJ provides methods that allow changes of parameters to occur only if the new set of schedulable objects is feasible. In these situations, the new parameters do not have an impact on a schedulable object's executions until its next release<sup>4</sup>. Some applications will need the changes to take place immediately (that is, to affect the current release). These unconditional changes are supported by the RTSJ through the methods that do not test for feasibility. Hence:

- methods which change parameters after feasibility analysis have an impact on the associated schedulable object at its next release;

---

<sup>4</sup>Although, it must be admitted that the current version of the RTSJ is unclear on which methods have an immediate impact and which methods have a deferred impact. The view expressed here is the authors' own interpretation of what is likely to be the situation in the next major release.

- methods which unconditionally change parameters have an immediate impact on the associated schedulable object's execution.

In both cases, the scheduler's feasibility set is updated. Of course, an infeasible system may still meet all its deadlines if the worst-case loading is not experienced (perhaps the worst-case phasing between the threads does not occur, or threads do not run to the worst-case execution time).

The only scheduler that the RTSJ fully defines is a priority scheduler, which can be summarized as having

**Scheduling policy.** The priority scheduler

- schedules schedulable objects according to their release profiles and scheduling parameters;
- supports the notion of base and active priority;
- orders the execution of schedulable objects on a single processor according to the active priority;
- supports a real-time priority range of at least 28 unique priorities (the larger the value, the higher the priority);
- requires the programmer to assign the base priorities (say, according to the relative deadline of the schedulable object);
- allows base priorities to be changed by the programmer at run time;
- supports priority inheritance or priority ceiling emulation inheritance for synchronized objects;
- assigns the active priority of a schedulable object to be the higher of its base priority and any priority it has inherited;
- deschedules schedulable objects when they overrun their CPU budgets;
- schedules event handlers for cost overrun and deadline miss conditions;
- enforces minimum inter-arrival time separation between releases of sporadic schedulable objects.

**Scheduling mechanism.** The priority scheduler

- supports pre-emptive priority-based dispatching of schedulable objects - the processor resource is always given to the highest priority runnable schedulable object;

- does not define where in the run queue (associated with the priority level) a pre-empted object is placed; however, a particular implementation is required to document its approach and the RTSJ recommends that it be placed at the front of the queue;
- places a blocked schedulable object that becomes runnable, or has its base priority changed, at the back of the run queue associated with its (new) active priority;
- places a schedulable object which performs a Thread.yield method call at the back of the run queue associated with its priority
- does not define whether schedulable objects of the same priority are scheduled in FIFO, round-robin order or any other order.

**Schedulability (feasibility) analysis.** The PriorityScheduler requires no particular analysis to be supported.

## 4 The Ada Model

The Ada model is well known to the workshop. Consequently, here it is simply summarised:

**Scheduling policy.** Ada

- supports the notion of base and active priority;
- orders the execution of tasks on a single processor according to the active priority;
- supports a real-time priority range of at least 32 unique priorities (the larger the value, the higher the priority);
- requires the programmer to assign the base priorities (say, according to the relative deadline of the task);
- allows base priorities to be changed by the programmer at run time;
- supports priority ceiling emulation inheritance for protected objects;
- assigns the active priority of a task to be the higher of its base priority and any priority it has inherited;

**Scheduling mechanism.** The priority scheduler

- supports pre-emptive priority-based dispatching of tasks – the processor resource is always given to the highest priority runnable task;

- places a pre-empted object at the front of its associated run queue;
- places a blocked task that becomes runnable, or has its base priority changed, at the back of the run queue associated with its (new) active priority;
- schedules tasks of the same priority in a FIFO manner.

**Schedulability (feasibility) analysis.** Ada assumes any analysis is off-line.

Ada provides lower level mechanisms which are invisible to the scheduler for handling release profiles (the delay statement) and deadline miss detection (the select then abort statement). Currently, there is no mechanism for budget enforcement or aperiodic server technology, although possible mechanisms are being proposed for future enhancements.

## 5 Conclusions

The goal of this paper was to argue that Java augmented with the next version of the RTSJ will become (once implementations are mature) a better real-time systems implementation technology than Ada. Section 2 argued that Suppositions 1,2 and 3 placed requirements on a real-time systems implementation technology to provide integrated support for a schedulable object's real-time characteristics. Consider again the suppositions.

### Supposition 1

*A real-time systems implementation technology should allow the programmer to express the characteristics of schedulable objects directly in the program and in a way that can be made available to the scheduler.*

The following characteristics have been identified.

**Release profile.** The RTSJ directly supports the notion of release profiles and makes them available to the scheduler. Ada only indirectly supports release profiles and they are invisible to the scheduler.

### Processing cost per release.

The RTSJ directly supports the notion of a CPU budget and integrates this support with the scheduler. Ada (currently) has no support in this area but may follow the POSIX approach and have CPU time clocks and timers. These would be invisible to the run-time scheduler.

### **Other hardware resources required per release.**

The RTSJ allows the memory requirements to be specified and exceptions to be thrown if they are violated. Ada provides some support in this area in the form of its storage pools.

### **Software resources required per release.**

The next release of the RTSJ is likely to support the notion of blocking time and make this visible to the run-time scheduler. Ada has no support in this area.

### **Deadline**

Deadlines are directly supported by RTSJ and integrated with the scheduling. Ada provides no direct support, and deadlines are invisible to the scheduler.

### **Value**

The RTSJ supports a simple integer number for value. Ada is silent on the issue.

## **Supposition 2**

*The scheduler must ensure that any deviation from a schedulable object's characteristics does not undermine any feasibility analysis that has been performed (off-line or on-line).*

The RTSJ deschedules any object which overruns its CPU budget and ensures that sporadic objects are not released closer than their minimum inter-arrival times. It monitors the amount of memory used and throws exceptions if maximum values are violated.

Ada provides no direct support. Any monitoring and response must be programmed at a lower level.

## **Supposition 3**

*The scheduler must provide mechanisms that can bound the impact of schedulable objects which have non-periodic release profiles.*

The RTSJ allows the impact of groups of schedulable objects to have a bounded demand for the processing resource. The scheduler enforces this.

Ada currently provides no mechanisms in this area.

As can be seen, in all areas the facilities provided by the RTSJ are more wide ranging and have a better integration with the run-time scheduler than the facilities provided by Ada. Hence, the paper concludes:

*Java augmented by the next version of the RTSJ will have better support for real-time systems development than Ada because it will have an integrated set of mechanisms for representing the characteristics of real-time activities and a framework from within which those activities can be scheduled. Ada 95, in contrast, has a set of orthogonal low-level mechanisms.*

Consequently, once efficient implementations of the full revised RTSJ emerge, and if Ada does not continue to evolve, Java augmented with the RTSJ will become a better real-time systems implementation technology than Ada 95. Ada must rise to the challenge and enhance its support for real-time systems in order to survive.

## **References**

- [1] G. Bollella, B. Brosgol, P. Dibble, S. Furr, J. Gosling, D. Hardin, and M. Turnbull. *The Real-Time Specification for Java*. Addison-Wesley, 2000.
- [2] A. Burns. The Ravenscar Profile. *ACM Ada Letters*, XIX(4):49–52, 1999.
- [3] A. Burns and A. J. Wellings. *Real-Time Systems and Programming Languages*. Addison Wesley, 3rd edition, 2001.
- [4] J. Kwon, A.J. Wellings, and S. King. Ravenscar-Java: A high integrity profile for real-time Java. In *Proceedings of the Joint ACM Java Grande - ISCOPE 2002 Conference*, pages 131–140, 2002.
- [5] TimeSys. JTime for Java: DataSheet. [http://www.timesys.com/files/prodlit/jtime-data\\_sheet.pdf](http://www.timesys.com/files/prodlit/jtime-data_sheet.pdf), Accessed 21/5/2003, 2003.
- [6] A. J. Wellings. *Concurrent and Real-Time Programming in Java*. Wiley, 2003.