# The Real-Time Specification for Java: Current Status and Future Work

Peter Dibble, TimeSys Corporation {peter.dibble@timesys.com}
Andy Wellings, University of York {andy@cs.york.ac.uk}

**ABSTRACT**

The Real-Time Specification for Java is now about two years old. It has been implemented, formed the basis for research and used in serious applications. Some strengths and weaknesses are becoming clear. This paper reviews the current status of the specification, outlines the challenges ahead and discusses areas where there is likely to be future design work.

## 1 Introduction

When Java emerged as a serious programming language in 1994, it was treated with disdain by much of the real-time community. Although the language was interesting from a number of perspectives — not least the fact that it had an integrated concurrent object-oriented programming model — the whole notion of Java as a real-time programming language was treated with skepticism. "Java and Real-time" was considered by many as an oxymoron! The last five years has seen a dramatic turnaround in the community's attitude. This has been brought about by the introduction of the Real-Time Specification for Java (RTSJ) [5,6] with its provision of real-time programming abstractions along with the promise of a predictable real-time virtual machine. Now, the newly formed real-time Java community argues over issues such as whether no-heap real-time threads are really necessary or whether real-time garbage collection combined with static analysis techniques can provide predictable response times in a hard real-time environment. There is little doubt that the real-time programming landscape has been irrevocably altered. However, there are still obstacles to be overcome before Java augmented by the RTSJ can challenge its main competitors in the real-time domain. This paper reviews some of the problems facing the real-time Java community over the next few years. The main challenges are in the following areas:

**Specification** — to produce a consistent and unambiguous Real-Time Specification for Java along with well-defined profiles for particular real-time application domains;

**Implementation** — to generate efficient implementations of real-time Java virtual machines (both open source and proprietary ones) for the full specification and the profiles;

**Maintaining Momentum** — to stimulate evolution of the specification in a controlled and sustained manner to add new functionality and to address new architectures.

This paper first presents a brief overview of the RTSJ and then considers each of the above topics in detail.

## 2 Background on the RTSJ

The RTSJ has several "guiding principles" which have shaped and constrained the development of the specification. These include requirements to

- be backward compatible with non real-time Java programs,
- support the principle of "Write Once, Run Anywhere" but not at the expense of predictability,
- address current real-time system practices and allow future implementations to include advanced features,
- give priority to predictable execution in all design trade-offs,
- require *no* syntactic extensions to the Java language,
- allow implementers flexibility.

The requirement for no syntactic enhancements to Java has had a strong impact on the manner in which the real-time facilities can be provided. In particular, all facilities have to be provided by an augmented virtual machine and a library of classes (and interfaces).

The RTSJ enhances Java in the following areas:

- memory management — a complementary model based on the concept of memory regions [24] (called scoped memory areas) allow the reclamation of objects without the vagaries of garbage collection;
- time values and clocks — high resolution time values and a real-time clock;
- schedulable objects and scheduling — pre-emptive priority based scheduling of real-time threads and asynchronous event handlers all within a framework allowing on-line feasibility analysis;
- asynchronous transfer of control — extensions to the Java thread interrupt mechanism to allowing the controlled delivery and handling of asynchronous exceptions;
- synchronization and resource sharing — support for the well-known priority inversion avoidance algorithms;

- physical and raw memory access — allowing more control over allocation of objects in different types of memory and interfacing to device drivers.

It should be stressed that the RTSJ is only intended to address the execution of real-time Java programs on a single processor systems. It attempts not to preclude execution on shared-memory multiprocessor systems but it has no facilities directly to control, for example, allocation of threads to processors.

# 3 Specification Challenges

A preliminary version of the RTSJ (version 0.9) was released in June 2000 [5], and then work commenced on a "reference implementation" (RI). Inevitably, the development and use of the RI uncovered errors and inconsistencies in the specification, some of which were removed in version 1.0 [6]. However, the prototype RI did not implement all features of the specification. As development of the RI continued, and researchers and other implementers began to experiment with and implement the full specification, so further inconsistencies and ambiguities in the 1.0 specification became apparent. Many of these have been removed in the 1.0.1 version that is due for release in the early part of 2004. However, some outstanding issues (whose resolution may require more significant changes) still need attention and are likely to be addressed in a subsequent version. Some of the main issues are outlined below.

## 3.1 Schedulable Objects

The RTSJ has made a good attempt to generalize real-time activities away from real-time threads towards the notion of schedulable objects. In the current specification both real-time threads and asynchronous event handlers are considered schedulable objects. Unfortunately, the operations that can be performed on a schedulable object are not consistently defined. For example, a schedulable object can create and enter into one or more scoped memory areas. The methods for manipulating the resulting scoped memory stack can be found partly in the `RealtimeThread` class and partly in the `AsyncEventHandler` class. Ideally, all operations associated with schedulable objects should be defined either in the `Schedulable` interface or in the class defining the associated functionality. In this example, the methods would be better defined either in the `Schedulable` interface or in the `MemoryArea` class.

## 3.2 Aperiodic and sporadic real-time threads

Although the RTSJ supports `AperiodicParameters` and `SporadicParameters` they can only successfully be used with asynchronous event handlers. There is no notion of a release event for an aperiodic (or sporadic) real-time thread and consequently it is difficult to see how an implementation can detect deadline miss or cost overrun. The resolution of this problem is tied closely to the resolution of the following issue.

## 3.3 The waitForNextPeriod method

This method is defined in the `RealtimeThread` class. However, it is only applicable to real-time threads with periodic release parameters. Ideally, the `RealtimeThread` class should contain a `waitForNextRelease` method rather than a `waitForNextPeriod` method. A `release` method would also be needed. For real-time threads with periodic parameters this method would be called by the implementation (although to be consistent with periodic events, any schedulable object would also be able to call it). This approach would also solve the previous problem with aperiodic threads, and provide a level of consistency between asynchronous event handlers and real-time threads (particularly in their approach to cost enforcement and deadline monitoring).

For further discussions on scheduling issues in the RTSJ see Wellings et al [28].

## 3.4 Rationale Time

Rational time is a relative time type that has an associated frequency. It is used to represent the rate at which certain events occur (for example, periodic thread execution). Hence a `RationalTime` value with, say, an interval of 1 second and a frequency of 100, has a inter-arrival time of 10 milliseconds..

`RationalTime` was a controversial class in version 1.0 of the RTSJ. The reasons for making it a subclass of `HighResolutionTime` were not clear and caused problems elsewhere in the specification (for example, the idea of passing a `RationalTime` as a deadline is very strange). For these reasons, the class has been deprecated in version 1.0.1 of the RTSJ and alternative approaches to meeting the original requirements will be provided in version 1.1.

# 4 Implementation Challenges

One of the key and immediate challenges facing the real-time Java community is to produce predictable and efficient implementations of the RTSJ. Three areas give particular cause for concern.

## 4.1 Memory Management

The RTSJ has established a new memory management model (via the introduction of memory areas) to facilitate more predictable memory allocation and deallocation. The result is that there are assignment rules that must be

obeyed by the programmer and that, consequently, must be checked by the implementation. Furthermore, the ability to nest memory areas means that illegal nesting must be prohibited. Whilst it is clear that static analysis of programs can eliminate many of the run-time checks [3,26], this requires special RTSJ-aware compilers or tools. The availability of these, along with efficient run-time implementations of the overall model, may well determine the eventual impact and take up of the RTSJ.

### 4.2 Asynchronous Transfer of Control

Introducing facilities for the management of ATC into Java was always going to be controversial [8]. However, there are undeniable real-time application requirements for such a facility [9]. It is a major challenge to keep the resulting overheads in the JVM small and predictable, and to ensure that code not using this facility suffers minimal impact.

### 4.3 Asynchronous Event Handling

The goal for asynchronous event handlers is to have a lightweight concurrency mechanism. Some implementations will, however, simply map an event handler to a real-time thread and the original motivations for event handlers will be lost. It is a major challenge to provide effective implementations which can cope with: heap-using and no-heap handlers, blocking and non-blocking handlers, daemon and non-daemon handlers, multiple schedulers, cost enforcement and deadline monitoring [13,29].

## 5 Maintaining Momentum

With the introduction of any new technology, there is a tension between producing a stable standard base that users can depend on, and providing a dynamic product that continues to evolve and address new application needs. So far, Java has caught the imagination of the user community, produced stable releases and maintained momentum for its evolution. If the RTSJ is to survive, it is important that it keeps pace with more general Java development and also that it develops its own momentum. This section reviews some of the major areas that may see future development.

### 5.1 The Ravenscar Profile

The Ravenscar[1] RTSJ profile [20, 17, 18] is probably the most sweeping potential extension to the RTSJ, but it is not strictly an extension. It uses some of the facilities of the RTSJ to specify a subset of the Java platform that might be suitable for safety-critical applications.

The most obvious feature of the Ravenscar Java proposal is that it does not permit garbage collection. All threads are trivially "no-heap." From the safety-critical point of view this is predictable, and it is much easier to implement a JVM without garbage collection than to implement one that lets some tasks instantly preempt garbage collection, but it could motivate a huge re-work of standard class libraries.

The RTSJ says nothing about the memory consumption of Java classes outside the RTSJ. It doesn't even say very much about memory consumption of classes in the RTSJ. The standard Java classes and the Java language itself are, however, designed to work with the garbage collector. Consider the Integer and String classes. All instances of those classes are immutable objects. In many cases that is a good property [4], but with no garbage collector immutable objects turn into memory leaks unless they are stored in scoped memory areas.

Or consider the ease with which a Java method creates an object and returns a reference to that object. In C or C++, the programmer would take care to track use of the returned object and delete it when it was no longer needed. More likely, the programmer would consider the pattern dangerous and avoid it altogether. In Java programs, garbage collection makes it safe and common. Without garbage collection the lifetimes of all these objects have to be considered, both in new applications and in the class libraries they use. This is, of course, the role of scoped memory. However, the scoped memory model has its limitations. Just as with standard Java it has been found necessary to allow interaction between the program and the garbage collector (via the introduction of "Reference" objects [22]), so it is necessary to allow interaction between the program and the implementation of scoped memory regions [7].

The proposed Ravenscar Java does much more than remove the garbage collector, but that one "simple" change is a far reaching.

That said, there is serious interest in safety critical Java, and something like the Ravenscar profile is likely to come about in the next few years. Indeed, recent efforts by the Open Group to form a new JSR to address this area attest to this.

### 5.2 Pluggable and Application-Level Schedulers

The vast majority of real-time operating systems support fixed priority pre-emptive scheduling with no on-line feasibility analysis. However, as more and more computers are being embedded in engineering (and other) applications, there is need for more flexible scheduling.

[1] The name is a historical reference that would be considered obscure outside the Ada community. Ravenscar is a village in the north of England where a safety-critical subset for Ada was proposed. That profile was implemented and used, and now will form part of the new Ada 2005 ISO standard. Since the proposed Java subset has much in common with the Ravenscar profile, that name has been informally (but firmly) attached to it.

Broadly speaking, there are three ways to achieve flexible scheduling [27]:

**Pluggable schedulers** — in this approach the system provides a framework into which different schedulers can be plugged. The CORBA Dynamic Scheduling [19] specification is an example of this approach. Kernel loadable schedulers also fall into this category.

**Application-defined schedulers** — in this approach, the system notifies the application every time an event occurs which requires a scheduling decision to be taken. The application then informs the system which thread should execute next. The proposed extensions to real-time POSIX support this approach [2].

**Implementation-defined schedulers** — in this approach, an implementation is allowed to define alternative schedulers. Typically this would require the underlying operating system or run-time support system (virtual machine, in the case of Java) to be modified. Ada 95 adopts this approach, albeit within the context of priority-based scheduling.

Support for new scheduling algorithms was a fundamental design goal of the RTSJ. Ideally it would have supported new pluggable schedulers from third parties and end users, but in the end the goal was to let JVM implementers add new schedulers. This approach allows the specification to depend on the scheduling implemented in the underlying operating system, and it lets the new scheduler use JVM APIs that are invisible to anyone but the implementer.

Unfortunately, this capability was not tested in the RI implementation process, but it has been exercised since that time. TimeSys has implemented several extensions of the `PriorityScheduler` class for their JTime [23] product. Those schedulers are an existence proof that the RTSJ supports a restricted form of pluggable schedulers.

These pluggable schedulers take every advantage of the implementer's control of system internals. They make extensive use of native methods to invoke normally inaccessible methods in the `PriorityScheduler`. There have even been cases where the `PriorityScheduler` had to be modified to invoke methods in its subclasses (a sign of poor design.) Hence, the specification does not provide a well-defined interface with which new schedulers can be easily added.

If pluggable schedulers were not a fundamental feature of the RTSJ, the inelegance of the existing approach to pluggable schedulers could be ignored, but they are a central feature of the RTSJ. Pluggable schedulers should allow researchers to add a scheduler without repeatedly inserting knowledge of that scheduler in the RTSJ class libraries. In the best case, the RTSJ might support some types of pluggable schedulers written by third parties (without source code for the JVM) or end users.

It will be hard to find a compatible way to improve the scheduler APIs for extension, but just as TimeSys' schedulers prove it can be done, they also prove that at least one vendor will go to great lengths to provide advanced schedulers. Researchers are also seriously interested in the possibilities of pluggable schedulers.

The Scheduler APIs need to be re-thought in the light of experience. Primarily this will take the form of new APIs for communication between the base `Scheduler` class and its subclasses, but the effort will probably spread to the `Parameter` classes. It remains an open issue as to whether these (or other) APIs should be made available to the applications programmer so that application-level schedulers can be defined.

## 5.3 Multiple Schedulers

The RTSJ supports multiple schedulers but falls short of specifying the interaction between concurrently active ones. Hence, the expected use case is that the application would choose a scheduler very early in its execution, make that the default, and use that scheduler exclusively for the body of the application's execution. This is probably a viable use case, but it isn't the only one. The fully expanded version of Jtime, for example, uses three cooperating schedulers: the priority scheduler, the high-priority scheduler, and the CPU-reservation scheduler.

The current RTSJ doesn't forbid cooperating schedulers, but it makes little effort to provide a useful framework for them. Support for multiple cooperating schedulers is not well-settled art and is still the subject of active research [12, 21, 1]. Since the right policy for interactions between multiple schedulers is still unknown, the RTSJ needs a flexible framework for multiple schedulers.

TimeSys did not have to address fundamental problems of multiple schedulers. All three of their schedulers are priority-based, and the interactions between the schedulers are supported by the underlying operating system. Ideally a framework for cooperating schedulers would:

- allow a thread under one scheduler to start a thread under another scheduler;
- supply APIs for a policy class that manages multi-level feasible sets where the system is partitioned in a flexible way among multiple schedulers; and
- supply APIs for a policy class that selects among the most eligible threads from each scheduler.

## 5.4 Cost Enforcement and Blocking

This area may be one of the most published shortcomings of the RTSJ. Cost enforcement is critical to many real-

time systems, and it is designed into the RTSJ as an optional feature. Unfortunately, the reference implementation did not choose to implement this feature and the specification was published with that aspect untested. With more years to consider cost enforcement, and the help of an implementation effort that includes cost enforcement [25], the specification's treatment of cost enforcement has been improved [28]. RTSJ version 1.0 was clear enough that it has been possible to get to a workable definition by interpreting the specification, but some areas will require extensions to the APIs that cannot be called "minor changes." For example, cost enforcement for sporadic real-time threads [28] needs further consideration.

Blocking is another area where the RTSJ is under specified. Currently, there is no notion of the time a schedulable object is blocked when accessing resources. This is needed for all forms of feasibility analysis. The solution to this problem could be as simple as introducing a blocking time term in the release parameters, or it could involve adding much more detailed information about all the resources used by each schedulable object.

## 5.5 Recycling Collections

Programmers who use the RTSJ soon discover that immortal memory is "viral." Objects placed in immortal memory tend to pull every associated object into immortal memory with them. This isn't a feature of the implementation. The force driving developers to locate all objects in immortal (or any single memory area) is convenience. References from immortal memory to scoped memory are illegal under the RTSJ assignment rules. References from scoped memory to immortal are legal, but the constraint to one-way references is sufficiently inconvenient that developers avoid the pain by putting whole families of objects in immortal.

This viral effect causes objects that should have limited lifetimes to be allocated in immortal memory. One solution to this problem is to extend the RTSJ to support weak scoped references (as already mentioned in Section 4.1), thereby allowing controlled references from immortal memory to scoped memory.

The other solution is to address the memory leak problem and to support a variation of a fixed block allocator that RTSJ users often call recycling lists. These keep a pool of pre-allocated objects in a memory area. The application gets them from the recycling list, uses them, and then returns them. There are some interesting problems. For instance, the JVM initializes objects when they are constructed and runs finalizers (if any) when objects are freed. Objects that live in a recycling list are constructed before they are placed in the list and might run finalizers if the list is ever freed. This is a minor issue for objects from some classes, but others classes might be impossible to recycle; for example, `Thread` objects are intended to go from construction to `start()` to termination. They can only be started once.

There is a rudimentary ancestor of the recycling list in RTSJ Platform Programming [13]. Recycling lists are not advanced computer science, but since everyone is implementing a version of the facility, it is time the specification standardized it.

## 5.6 Scoped Memory Subclasses

There is much demand for extensions to the scoped memory classes. Unfortunately, useful extensions here require help from the JVM. There are probably as many ideas for useful new scoped memory classes as there are RTSJ users. For instance:

- A reference-counting scope that allows an object's space to be reclaimed when there are no references to it, even though the scoped memory is still in use.
- A storage scope that can retain objects when its reference count goes to zero.
- A scope that permits malloc- and free-like operations

The RTSJ could approach the need for more types of scoped memory by adding some new scoped memory classes, or it could update the memory area specification so subclasses have a way to control the operation of the base classes.

Of course, static analysis techniques might allow some objects to be created on the stack and, therefore, obviate the need for explicit scoped memory coding [3, 26]. If this can be sufficiently generalized, and if real-time garbage collection becomes more efficient and accepted, one of the major features of the RTSJ may need deletion, or it might live on as a feature that is heavily used, but not by programmers.

## 5.7 Multiprocessor and Distributed Systems

The RTSJ ignores issues related to all types of multi-processor systems. JSR 50 [15] is addressing the distributed systems but progress has been slow and results are few [30]. However, there may be room for a comparatively unsophisticated approximation to distributed real-time Java under the RTSJ. Support for real-time processing on SMP and NUMA platforms is not addressed by the RTSJ or another JSR, but a fair number of RTSJ users are running it on SMP systems. Perhaps the scheduling techniques that make this work without real-time anomalies should be added to the specification.

## 5.8 Multiple Clocks

The 1.0.1 version of the RTSJ clarifies the treatment of multiple clocks under the RTSJ, but it does not specify the

behavior of any clocks other than the default real-time clock. It is probably inappropriate to *require* any other clocks, but it would be beneficial to define the APIs and behaviors of several additional clocks.

A *consumed CPU time* clock would be particularly useful. It is an anomaly that the RTSJ programmer can request that a cost overrun handler can be released when a schedulable object consumes more CPU time than in its stated budget; yet the programmer cannot determine how much CPU time the schedulable object has currently consumed. A consumed CPU time clock would also augment the processing group facilities of the RTSJ and allow more flexible bandwidth preserving algorithms to be implemented, including sporadic servers.

### 5.9 Multiple Criticalities

The RTSJ includes an `ImportanceParameters` class, but does not specify the behavior of a scheduler that employs them. Importance could be used to order queues within priority, they could be combined with data from release parameters to select the least harmful set of deadlines to miss in case of overload, and there are probably endless other requirements for this two-dimensional class of scheduling parameters. It would be productive to specify at least one scheduler that could use `ImportanceParameters`.

There are related scheduling mechanisms that the RTSJ does not touch. For instance ARINC 651 and its associated standards ARINC 653 (APEX) and ARINC 659 [11] support the notations of partitions and processes within a partition. Scheduling divides each interval of time into quanta that are then allocated to partitions. Each process is executable only within the time allocated to its owning partition (using priority-based scheduling). This simple mechanism effectively isolates the CPU consumption of software components. This prevents overload in a component from degrading the entire system and makes integration and testing of complex systems comparatively simple. Of course, the tradeoff is the potential increase in release jitter and more complex feasibility analysis.

Java is already considering the notion of an isolate [16] where more than one application can share access to the same JVM. There has been some work that is taking this approach and integrating it into Ravenscar-Java [10]. Here, processing group parameters define the scheduling at the application level, and the RTSJ priority scheduler is used to schedule within applications. It would be interesting to generalize this and, for example, include an optional ARINC 653 scheduler in the RTSJ.

### 5.10 Alternative Interrupt Handling Models

The RTSJ handles external events such as signals and hardware interrupts by raising *happenings*. Happenings can be associated with asynchronous events and thus interrupt handlers become asynchronous event handlers. Whilst this provides the flexibility to schedule interrupt handlers in competition with other activities, some consider the approach too expensive.

It is hard to implement the flexibility provided by the RTSJ approach without sacrificing performance. An asynchronous event handler can increment a counter, release a lock, fire another event, or anything else that can be coded under RTSJ. However, other approaches exist that can perform operations like these directly and much more efficiently than firing an asynchronous event that starts an asynchronous event handler that performs the actions. This is not to suggest that the current mechanism for handling happenings should be removed, but some additional mechanism that could do a few simple things very efficiently would benefit the more performance-oriented developers. For example, Ada allows the equivalent of a synchronized method to be called directly in response to an external happening.

## 6 Conclusions

This paper has reviewed the current status of the RTSJ, outlined the challenges it faces and proposed some areas for future work. Many of the extensions have ambitious goals; some would make interesting research projects. Since the Java Community Process [14] governs the evolution of the RTSJ, it will deliberate and will weight compatibility heavily when considering extensions, but derivative works and non-standard extensions to the RTSJ are not so constrained.

## Acknowledgements

The authors gratefully acknowledge the many discussions held with the other primary authors of the Version 1.0.1 RTSJ in particular, Rudy Belliardi, Ben Brosgol, and David Holmes. Those discussions have led to some of the ideas expressed in this paper.

## References

[1] Abeni, L. and Buttazzo, G., “Hierarchical QoS Management for Time Sensitive Applications”, Proceedings of the IEEE Real-Time Technology and Applications Symposium, 2001.

[2] Aldea Rivas, M., and González Harbour, M., “POSIX-Compatible Application-Defined Scheduling in MaRTE OS”, Proceedings of 14th Euromicro Conference on Real-Time Systems, Vienna, Austria, IEEE Computer Society Press, pp. 67-75, 2002.

[3] Beebee W., and Rinard, Martin, “An Implementation of Scoped Memory for Real-Time Java”, *Proceedings of Embedded Software, First International Workshop, EMSOFT 2001,* Tahoe City, California October 2001.

[4] Bloch, J., *Effective Java Programming Language Guide*, Addison Wesley, 2001.

[5] Bollella, G., Brosgol, B., Dibble, P., Furr, S., Gosling, J., Hardin, D., and Turnbull, M., *The Real-Time Specification for Java (version 0.9)*, Addison Wesley, 2000.

[6] Bollella, G., Brosgol, B., Dibble, P., Furr, S., Gosling, J., Hardin, D., Turnbull, M., and Belliardi, R., *The Real-Time Specification for Java (1.0)(2001)*, available from www.rtj.org.

[7] Borg, A. and Wellings, A.J., "Reference Objects for RTSJ Memory Areas", On the Move to Meaningful Internet Systems 2003: Workshop on Java Technologies for Real-Time and Embedded Systems, LNCS, volume 2889, pp 397-340}, Springer, 2003.

[8] Brosgol, B., and Wellings, A.J., "A Comparison of Asynchronous Transfer of Control Features in Ada and the Real-Time Specification for Java", Reliable Software Technologies – Ada Europe, volume 265, Lecture Notes in Computer Science, pp 113-128, Springer-Verlag, 2003.

[9] Burns, A. and Wellings, A.J., *Real-Time Systems and Programming Languages*, 3rd Edition, 2001, Addison Wesley.

[10] Cai, H, and Wellings, A.J., "A Real-Time Isolate Specification for Ravenscar-Java", Proceedings of the Seventh International IEEE Symposium on Object-Oriented Real-Time Distributed Computing, ISORC 2004.

[11] Carpenter K.H.T, Driscoll K., and Carciofini J., "ARINC 659 Scheduling: Problem Definition", *Proceedings of IEEE Real-Time Systems Symposium*, pp 165–169, 1994.

[12] Deng, Z. and Liu, J. W.-S. , "Scheduling Real-Time Applications in an Open Environment," Proceedings of the IEEE Real-Time Systems Symposium, IEEE Computer Society Press, pp 308-319, 1997.

[13] Dibble, P., *Real-Time Java Platform Programming*, Prentice Hall, 2001.

[14] Java Community Process, Home Page, <http://www.jcp.org>.

[15] Java Community Process, JSR 50, "Distributed Real-Time Specification", http://www.jcp.org/jsr/detail/50.jsp, 2000.

[16] Java Community Process, JSR121, "JSR 121: Application Isolation API Specification", http://www.jcp.org/jsr/detail/121.jsp, 2001.

[17] Kwon J., Wellings, A.J., and King, S., "Assessment of the Java Programming Language for Use in High Integrity Systems", University of York Computer Science Technical Report 2002341, 2002.

[18] Kwon J., Wellings, A.J., and King, S., "Ravenscar-Java: A High Integrity Profile for Real-Time Java", Proceedings of the Joint ACM Java Grande - ISCOPE 2002 Conference pp 131-140, 2002.

[19] OMG, "Real-time Corba 2.0 Dynamic Scheduling Specification", OMG Document orbos/ 01-08-34, 2001.

[20] Puschner, P., and Wellings, A.J., "A Profile for High Integrity Real-Time Java Programs", Proceedings of the 4th IEEE Symposium on Object-Oriented Real-Time Distributed Computing, ISORC, pp 15-22, 2001.

[21] Regehr, J. and Stankovic, J.A., "HLS: A Framework for Composing Soft Real-Time Schedulers, Proceedings of the IEEE Real-Time Systems Symposium, pp 3-14, 2001.

[22] Sun Microsystems, "The Reference Object API for Sun JDK 1.3", available at http://java.sun.com/j2se/1.3/docs/guide/refobs.

[23] TimeSys, JTime, http://www.timesys.com/index.cfm?bdy=java_bdy.cfm.

[24] Tofte M. and Talpin, J., "Region-based Memory Management", Information and Computation, 132(2) pp 109-167, 1997.

[25] The OVM Project, http:www.ovmj.org.

[26] Salcianu, A., and Rinard, M., "Pointer and Escape Analysis for Multithreaded Programs", ACM SIGPLAN Notices, 36.7, pages 12–23, 2001.

[27] Wellings, A.J., *Concurrent and Real-Time Programming in Java*, Wiley, 2004.

[28] Wellings, A.J., Bollella, G, Dibble, P, and Holmes, D., "Cost Enforcement and Deadline Monitoring in the Real-Time Specification for Java", Proceedings of the Seventh International IEEE Symposium on Object-Oriented Real-Time Distributed Computing, ISORC 2004.

[29] Wellings, A.J. and Burns, A., "Asynchronous Event Handling and Real-time Threads in the Real-Time Specification for Java", Proceedings of the 8th IEEE Real-Time and Embedded Technology and Applications Symposium, pp 81-8, 2002.

[30] Wellings, A.J., Clark, R., Jenson, D. and Wells, D., "A Framework for Integrating the Real-Time Specification for Java and Java's Remote Method Invocation", Proceedings of the Fifth International IEEE Symposium on Object-Oriented Real-time Distributed Computing ISORC 2002.