Java for Safety-Critical Applications

Thomas Henties[1]

Siemens AG

James J. Hunt

aicas

Doug Locke

Locke Consulting, LLC

Kelvin Nilsen

Aonix NA

Martin Schoeberl

Institute of Computer Engineering
Vienna University of Technology

Jan Vitek

Computer Science Dept.
Purdue University

Abstract

In recent years, various approaches to real-time execution of Java have proven their worth in numerous commercial and defense applications. The Real-time Specification for Java has extended the Java platform with a range of features needed for real-time computing. As the use of real-time Java has become more widespread, the demand for Java in real-time applications with safety requirements has led to an effort to define a new standard—JSR-302 Safety-Critical Java (SCJ). The goal of this standard is to facilitate the creation of safety-critical Java applications capable of certification under standards such as DO 178B level A or IEC61508 for SIL 4. JSR-302 is nearing completion and will soon be released for public review. This paper introduces some of the primary goals, challenges, and proposed solutions for safety-critical Java and its relationship with the Real-time Specification for Java.

Keywords: Real-Time, Java, Safety-Critical Systems

[1] Email: thomas.henties@siemens.com

Fig. 1. A sample Real-time Java application: the first integrated real-time Java system on the ScanEagle Unmanned Aerial Vehicle [9]. The flight hardware/software was an Embedded Planet PowerPC 8260 processor running at 300MHz with 256 MB SDRAM and 32 MB FLASH. The operating system is Embedded Linux.

1 Introduction

Most microprocessors are now used for real-time or embedded applications, and the behavior of many of these applications is constrained by the physical world. Higher-level programming languages and middleware are needed to robustly and productively design, implement, compose, integrate, validate, and enforce both real-time constraints and conventional functional requirements, while assuring modularity and composability of independently developed components. It is essential that the production of real-time embedded systems can take advantage of languages, tools, and methods that enable higher software productivity. The Java programming language has become an attractive choice because of its safety, productivity, its relatively low maintenance costs, and the availability of well trained developers.

Although it features good software engineering characteristics, Standard Edition Java is unsuitable for real-time embedded systems due to under specification of thread scheduling, synchronization, and garbage collection. By imprecisely specifying the semantics of these JVM services, the designers of Standard Edition Java made it possible to easily port Java to a broad diversity of platforms. At the same time, the Java language specification does not preclude Java virtual machine vendors from providing implementations that support more reliable compliance with real-time constraints. The Real-Time Specification for Java (RTSJ) [12] represents a standardized approach to enhancing the Java virtual machine by tightening the semantics of thread scheduling and synchronization and providing mechanisms that allow real-time Java programs to run without interference from garbage collection. The RTSJ, along with various vendor-specific enhancements to the Java platform, have enabled development of large-scale real-time embedded systems in the Java language [28,15,20]. Real-time Java has established itself as a viable alternative for developing large real-time systems as evidenced by the development of open source and commercial virtual machines such as Mackinac [11], WebSphere Real Time [10], PERC [7], and JamaicaVM [6]. Recently, Lockheed Martin teamed with Aonix to modernize the Aegis cruiser fleet using the PERC virtual machine [4] while IBM and Raytheon have teamed to do build the battleship computing environment software for the new DDG-1000 warship using IBM's WebSphere Real Time product [19]. Other notable applications include unmanned aircraft [8,1,3], audio processing [17], industrial automation [16], railway automation and supervision [25], and flight entertainment systems [5].

Safety-critical systems are systems in which an incorrect response or an incorrectly timed response can result in significant loss to its users; in the most extreme case, loss of life may result from such failures. A secure system, on the other hand, is one in which mechanisms are in place that protect the information within the system from theft or corruption. Secure systems often control access to the system with security policies. The focus of the Safety Critical Specification for Java is on safety-critical applications rather than system security, but many of the mechanisms needed for safety-critical systems are closely related to those needed for system security.

For this reason, safety-critical applications require an exceedingly rigorous validation and certification process. Such certification processes are often required by legal statute or by certification authorities. For example, in the United States, the Federal Aviation Administration requires that safety-critical software be certified using the Software Considerations in Airborne Systems and Equipment Certification (DO-178B [22]). European aviation regulators require certification under essentially identical ED-12B [23] standard. DO-178B and ED-12B where developed jointly under the auspices of the Radio Technical Commission for Aeronautics (RTCA) and the European Organisation for Civil Aviation Equipment (EUROCAE).

The Safety-Critical Java (SCJ) specification is designed to enable the creation of safety-critical applications using a safety-critical Java infrastructure and using safety-critical libraries that are amenable to certification under DO-178B, Level A and other safety-critical standards. In the context of Java technology, this means a much tighter and smaller set of Java virtual machines and libraries, and much more precise performance requirements on the virtual machines and libraries. Additionally, the applications must exhibit freedom from exceptions that cannot be successfully handled. This requires, for example, that there be no memory access errors at runtime.

The specification is being developed within the framework of the Java Community Process (JCP) which allows individuals and organizations to evolve definition of the Java platform. Besides a specification, a Reference Implementation (RI) is being developed to provide a proof of feasibility. Furthermore a test suite, referred to as a Technology Compatibility Kit (TCK) will ensure that independent implementations of the specification are compatible. The TCK and RI complement the specification and are mandatory under the JCP rules. The final specification will require approval by the JCP's Executive Committee.

2 Definitions and Background

In a real-time system, beside the calculation of correct results, it is crucial that these result are calculated within well defined time bounds; neither too early nor too late. Depending on the consequences of missing a deadline, real time systems can be classified into two categories.

- *Soft real-time* when missing a deadline will lead to a loss of quality. Examples are: decoding of a video stream or a telephone exchange.

- *Hard real-time* if missing a deadline will lead to system failure. Examples include the control of vital vehicle functions such as brakes, aileron control, and robot control.

It is important to recognize that classifying a system as hard real-time does not imply that every operation in the system is a hard real-time operation, but rather it means that some operations must be handled as hard real-time.

Safety-critical systems are systems in which incorrect behavior may cause loss of human life or a severe injury. Often, a safety-critical system's behavior includes not only its functionality, but also its timeliness. In other words, many safety-critical systems are also hard real-time systems. Examples can be found in the fields of aviation, military, medicine, and power plant controls.

There are presently a number of tools that statically analyze several key properties of Java programs or class files respectively. The SCJ standard assumes that such tools will be provided by SCJ implementations, so they are intentionally not part of SCJ. The standard is intended to encourage further development, enhancements and innovation. Data flow analysis, model checking, and deductive verification are potential approaches to guarantee key safety properties such as post conditions, reachability, or worst-case execution time. A simple example for such a static analysis is a proof that the initialization sequence for a set of classes contains no cycle. It is explicitly stated, that neither all applications written in SCJ are inherently certifiable nor that all desirable static analysis can be applied to every SCJ-application. In fact, SCJ facilitates both as far as possible and it is expected that vendors will provide appropriate tools. Remark that the analysis of SCJ is far more easy than a language like C. If necessary, tools may use Java annotations to allow users to provide additional information to easy analysis.

2.1 Real-time Specification For Java

The Real-Time Specification for Java (RTSJ) was developed within the Java Community Process as the first Java Specification Request (JSR-1). Its goal was to "provide an Application Programming Interface that will enable the creation, verification, analysis, execution, and management of Java threads whose correctness conditions include timeliness constraints" [12] through a combination of additional class libraries, strengthened constraints on the behavior of the JVM, and additional semantics for some language features, but without requiring special source code compilation tools. The RTSJ covers five main areas related to real-time programming.

- *Scheduling*: Priority based scheduling guarantees that the highest-priority schedulable object is always the one that is running (in a single processor application). The scheduler must also support the periodic release of real-time threads, and the sporadic release of asynchronous event handlers that can be attached to asynchronous event objects that themselves are triggered by actual events in the execution environment.
- *Admission control and cost enforcement*: Schedulable objects can be assigned parameter objects that characterize their temporal requirements in terms of start times, deadlines, periods, and cost. This information can be used to prevent the

admission of a schedulable object if the resulting system would not be feasible from a scheduling perspective. Schedulable objects can also have handlers that are released in the event of a deadline miss.

- *Synchronization*: Priority inversion through the use of Java's synchronization mechanism (monitors) are controlled by using the priority inheritance protocol (PIP), or optionally, the priority ceiling emulation protocol (PCEP). This applies to both application code and the virtual machine itself.
- *Memory Management*: Time-critical threads must not be subject to delays caused by garbage collection. To facilitate this, a `NoHeapRealtimeThread` is prohibited from touching heap allocated objects, and so can preempt garbage collection at any time. Instead of using heap memory, these threads can use special, limited-lifetime memory areas known as *scoped memory areas*, or an immortal memory area from which objects need never be reclaimed.
- *Asynchronous Transfer of Control*: It is sometimes desirable to terminate a computation at an arbitrary point. The RTSJ allows for the asynchronous interruption of methods that are marked as allowing asynchronous interruption [13]. This facilitates early termination while preserving the safety of code that does not expect such interruptions.

2.2 Limitations of SCJ with respect to the RTSJ

The main topic of JSR-302 is to facilitate a certification (e.g. DO 178B level A) of Java programs as far as possible. For this purpose, radical subsetting of the full Java environment has been required.

First of all, because garbage collection is not supported, heap memory is not available, so some convenient methods of the `java.lang` package can not be supported. Emphasis is placed on using periodic event handlers instead of threads. Concerning threads, only `NoHeapRealtimeThread` can be used, and then only under stringent conditions. Both event handlers and threads use preemptive, priority based scheduling. SCJ supports the priority ceiling emulation protocol for avoiding priority inversion, but not the priority inheritance protocol.

The object oriented programming style forced by the Java programming language requires dynamic allocation of temporary objects. This is facilitated by stack based scoped memory. Thus garbage collection is not needed and tools can prove that there are no dangling pointers to released objects.

3 Safety Critical Java

The RTSJ imposes few limitations on how a developer structures an application, and supports a wide variety of software models in terms of concurrency, packaging, synchronization, memory, etc. Because safety-critical applications must generally conform to rigorous certification requirements, they generally use much simpler programming models that are amenable to certification. SCJ defines a more limited programming model. This is accomplished by defining concepts such as missions, limited startup procedures, and levels of compliance. In addition, special annotations are defined that are intended for use by vendor-supplied and/or third-party tools to perform static off-line analysis that can ensure many critical correctness properties for the safety-critical application.

3.1 The Mission Concept

An SCJ compliant application will consist of one or more *missions.* A mission consists of a bounded set of periodic event handlers and possibly some instances of `NoHeapRealtimeThread`, known collectively as *Schedulable Objects*, a term defined by the RTSJ. For each mission, a dedicated block of memory is identified as the mission memory. Objects created in mission memory persist until the mission is terminated, and their resources will not be reclaimed until the mission is terminated. All classes are loaded into immortal memory when the system starts up. Conforming implementations are not required to support dynamic class loading. There is no requirement that classes, once loaded, must ever be removed, nor that their resources be reclaimed. Each mission starts in an initialization mode during which objects may be allocated in mission memory. When a mission's initialization has completed, mission mode is entered. During mission mode, objects must not be created in immortal memory or mission memory, but mutable objects residing in mission or immortal memory may be modified as needed. All application processing for a mission occurs in one or more schedulable objects. When a schedulable object is started, its initial memory area is a scoped memory area that is entered when the schedulable object is released and is exited (and the memory area cleared) when the release terminates. During mission mode, objects cannot be created in immortal memory or mission memory, but mutable objects residing in these memory areas may be modified as needed. Objects allocated during mission execution will be allocated in scoped memory. This scoped memory area is not shared with other schedulable objects. If an application tries to create a new instance in a forbidden area an `InacessibleAreaException` will be thrown. (However, in a safety critical application it is desirable that the absence of such an exception can be proved a priori - but this is out of the scope of the language specification.) The mission framework provides a mechanism for orderly termination. Once termination is requested, all schedulable objects in the mission are notified to cease operating. Once they have all stopped computation, the mission can run cleanup code before it terminates. This provides a clean restart or transition to a new mission.

3.2 Compliance Levels

The complexity of safety-critical software varies greatly. At one end of the spectrum, safety-critical applications contain only a single thread and support only a single function, with only simple timing constraints. At the other end, there exist highly complex multi-modal safety-critical systems. The cost of certification of both the application and the infrastructure is highly sensitive to their complexity, so enabling the construction of simpler applications and infrastructures is highly desirable. Therefore, SCJ defines three compliance levels to which both implementations and applications may conform. The SCJ refers to the distinct compliance configurations as Level 0, Level 1, and Level 2. Level 0 refers to the simplest applications and Level 2 refers to the more complex applications. The use of a sequence

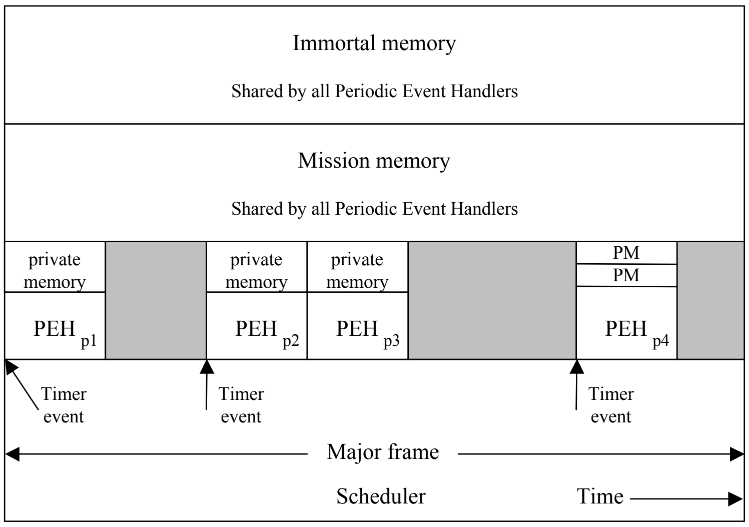

Fig. 2. Timeline of a Level 0 application.

of numbers to denote these levels is intended to reinforce the idea that these compliance levels can be considered nested. For example, it is required that applications written to run at each level must be upward compatible to higher levels.

3.2.1 Level 0

A Level 0 application's programming model is a familiar model often described as a timeline model, a frame-based model, or a cyclic executive model. In this model, the mission can be thought of as a set of computations, each of which is executed periodically in a precise, clock-driven timeline, and processed repetitively throughout the mission. Figure 2 illustrates a simple Level 0 application.

A Level 0 application's schedulable objects shall consist only of a set of Periodic Event Handlers (PEHs). Each PEH has a period, priority, and start time relative to the beginning of a major cycle. A schedule of all PEHs is constructed by either the application designer or by an offline tool provided with the implementation.

All PEHs execute under control of a single underlying thread. This enforces the sequentiality of every PEH, so the implementation can safely ignore synchronization in the application. The application developer, however, is strongly encouraged to include the synchronization required to safely support its shared objects so the application can run correctly on a Level 1 or Level 2 implementation as well. The use of a single thread to run all PEHs without synchronization implies that a Level 0 application runs only on a single CPU. If more than one CPU is present, it is necessary that the state managed by a Level 0 application not be shared by any application running on another CPU. The operations `wait` and `notify` are not available at Level 0. Each PEH has a private scoped memory area created for it before invocation that will be entered and exited at each invocation. It is important to note that an operation which blocks, whether inside a synchronized method or otherwise, will be blocking the entire application if it is running on a Level 0 implementation.

3.2.2 Level 1

A Level 1 application uses a familiar programming model consisting of a single mission with a set of concurrent computations, each with a priority, running under control of a fixed-priority preemptive scheduler. The computation is performed in a set of PEHs and/or Aperiodic Event Handlers (APEHs). A Level 1 application shares objects in mission memory among its PEHs and Aperiodic Event Handlers (APEHs), using synchronized methods to maintain the integrity of its shared objects. Each Level 1 PEH or APEH has a private scoped memory area created for it before invocation that will be entered and exited at each invocation. During execution, the PEH or APEH may create, enter, and exit one or more other scoped memory areas, but these scoped memory areas must not be shared among PEHs or APEHs. The operations `wait` and `notify` are not available at Level 1.

3.2.3 Level 2

A Level 2 application starts with a single mission, but may create and execute additional missions concurrently with the initial mission. Computation in a Level 2 mission is performed in a set of PEHs, APEHs, and/or `NoHeapRealtimeThread`. Each child mission has its own mission sequencer, its own mission memory, and may also create and execute other child missions. Each Level 2 PEH, APEH, or `NoHeapRealtimeThread` has a private scoped memory area created by the runtime system for it before invocation. For PEHs and APEHs, the private scoped memory area will be entered and exited at each invocation. For a `NoHeapRealtimeThread`, the private scoped memory area will be entered when it starts its `run` method and exited when the `run` method terminates. During execution, each PEH, APEH, or `NoHeapRealtimeThread` may create, enter, and/or exit one or more other scoped memory areas, but these scoped memory areas must not be shared among PEHs or APEHs. A Level 2 application may use `wait` and `notify` operations.

3.3 Use of Asynchronous Event Handlers

The RTSJ defines two mechanisms for real-time execution: the `RealtimeThread` class, which uses a style similar to `java.lang.Thread` for concurrent programming, and the `AsynchEventHandler` class, which is event based. To facilitate analyzability, this specification provides only `AsynchEventHandler` at Levels 0 and 1, permitting `NoHeapRealtimeThread` only at Level 2.

4 Implementations

Although the specification is in the draft phase, James Hunt has implemented an initial version of the Reference Implementation (RI) to verify that the specification is suitable for the safety-critical domain. The RI is intended to run on top of a standard RTSJ implementation. The RI with the underlying RTSJ is not intended to be used in a product that will be safety certified. The intention of the RI is to provide a reference for future implementations. The RI and the TCK will be provided under an open-source license. Therefore, it is expected that several commercial and academic implementations of SCJ will follow.

The product PERC Pico from Aonix [7], already in use for hard real-time Java systems, is expected to be adapted to conform to the SCJ. Some of the library annotations of SCJ are based on the annotations from PERC Pico.

A different approach to real-time Java is taken by JOP [26], a time predictable Java processor. The processor is designed to simplify worst-case execution time analysis and executes Java bytecodes as native language. The current runtime environment for JOP consists of a real-time thread definition that is similar to SCJ level 1. A layer to map the SCJ API to that layer is currently under development. When the SCJ specification is finalized, SCJ will be the primary API for real-time Java applications on JOP. SCJ running on JOP is used as a platform for real-time Java development on multicore processors within the EU project Jeopard [29].

5 Related Work

The SCJ builds upon a broad research base on using Java (and Ada) for hard real-time systems, sometimes also called high integrity systems. The Ravenscar profile defines a subset of Ada to support development of safety-critical systems [14]. Based on the concepts of Ravenscar Ada a restriction of the RTSJ was first proposed in [21]. These restrictions are similar to SCJ level 1 without the mission concept. The idea was further extended in [18] and named the Ravenscar Java profile (RJ). RJ is an extended subset of RTSJ that removes features considered unsafe for high integrity systems. Another profile for safety-critical systems was proposed within the EU project HIJA [2].

PERC Pico from Aonix [7] defines a Java environment for hard real-time systems. PERC Pico defines its own class hierarchy for real-time Java classes which are based on the RTSJ libraries, but are not a strict subset thereof. PERC Pico introduces stack-allocated scopes, an elaborate annotation system, and an integrated static analysis system to verify scope safety and analyze memory and CPU time resource requirements for hard real-time software components. Some of the annotations used to describe the libraries of the SCJ are derived indirectly from the annotation system used in PERC Pico.

Another definition of a profile for safety-critical Java was published in [27]. In contrast to RJ the authors of that profile argue for new classes instead of reusing RTSJ based classes to avoid inheriting unsafe RTSJ features and to simplify the class hierarchy. A proposal for mission modes within the former profile [24] permits recycling CPU time budgets for different phases of the application. Compared to the mission concept of SCJ that proposal allows periodic threads to vote against the shutdown of a mission. The concept of mission memory is not part of that proposal.

6 Conclusion

Java, as a strongly typed, object oriented language, detects common programming errors at compile time. Exception handling, threads and synchronization are part of the language. Together with runtime checks of array bounds, reference errors that can lead to a crash of the program are avoided in Java. Therefore, Java is an interesting language for building safety-critical application.

However, standard Java and the RTSJ are too large and complex for safety-critical certification. Therefore, the SCJ defines a subset of the RTSJ intended for safety-critical applications. The scoped memory model of the RTSJ is restricted to allow static analysis of the memory usage. The thread model is largely restricted to periodic and asynchronous event handlers to simplify the schedulability analysis. The concept of missions and sub-missions at higher levels reintroduces dynamic features of the RTSJ in a safer form.

The potential for tools to prove properties is much higher than for C or C++, because the language semantics are defined with higher precision and with less ambiguity. Java applications can be analyzed at the bytecode level that is the standardized intermediate representation of Java applications. As soon as the certifying authorities accept JVMs that implement SCJ, the use of Java will lead to higher productivity in the development of safety-critical applications. Beside the already mentioned advantages of Java, the demanded software quality can be verified more easily.

Acknowledgement

The SCJ specification is a collaborative work of the Expert Group for SCJ. The authors thank the active members of the expert group: B. Scott Andersen, Ben Brosgol, Mike Fulton, Johan Nielsen, Joyce Tokar, and Andy Wellings.

References

[1] *Boeing selects software for j-ucas x-45c*, Defense Industry Daily (2005).
URL `http://www.defenseindustrydail.com/boeing-selects-software-for-jucas-x45c-01413/`

[2] *Hija safety critical java proposal*, available at http://www.aicas.com/papers/scj.pdf (2006).

[3] *The jamaicavm brings java technology to mission software in an unmanned aircraft by eads*, Military Embedded Systems (2006).
URL `http://www.mil-embedded.com/news/db/?3302`

[4] *Lockheed martin selects aonix perc virtual machine for aegis weapon system*, Military Embedded Systems (2006).
URL `http://www.mil-embedded.com/news/db/?4224`

[5] *Aonix perc selected for inflight entertainment system*, Embedded Computing Design (2007).
URL `http://www.embedded-computing.com/news/db/?8205`

[6] aicas, *The Jamaica Virtual Machine homepage,* `http://www.aicas.com` (2005).

[7] Aonix, *Perc pico 1.1 user manual*, http://research.aonix.com/jsc/pico-manual.4-19-08.pdf (2008).

[8] Armbruster, A., J. Baker, A. Cunei, C. Flack, D. Holmes, F. Pizlo, E. Pla, M. Prochazka and J. Vitek, *A real-time java virtual machine with applications in avionics*, ACM Transactions on Embedded Computing Systems (TECS) **7** (2007).

[9] Armbuster, A., J. Baker, A. Cunei, D. Holmes, C. Flack, F. Pizlo, E. Pla, M. Prochazka and J. Vitek, *A Real-time Java virtual machine with applications in avionics*, ACM Transactions in Embedded Computing Systems (TECS) (2006).

[10] Auerbach, J., D. F. Bacon, B. Blainey, P. Cheng, M. Dawson, M. Fulton, D. Grove, D. Hart and M. Stoodley, *Design and implementation of a comprehensive real-time Java virtual machine*, in: *Proceedings of the 7th ACM & IEEE international conference on Embedded software (EMSOFT)*, 2007, pp. 249–258.

[11] Bollella, G., B. Delsart, R. Guider, C. Lizzi and F. Parain, *Mackinac: Making hotspot real-time*, in: *Proceedings of the Eighth IEEE International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC'05)*, 2005, pp. 45–54.

[12] Bollella, G., J. Gosling, B. Brosgol, P. Dibble, S. Furr and M. Turnbull, "The Real-Time Specification for Java," Addison-Wesley, 2000.

[13] Brosgol, B., S. Robbins and R. Hassan II, *Asynchronous transfer of control in the Real-Time Specification for Java*, in: *Proceedings of the Fifth International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC)*, 2002.

[14] Dobbing, B. and A. Burns, *The ravenscar tasking profile for high integrity real-time programs*, in: *Proceedings of the 1998 annual ACM SIGAda international conference on Ada* (1998), pp. 1–6.

[15] Dvorak, D., G. Bollella, T. Canham, V. Carson, V. Champlin, B. Giovannoni, M. Indictor, K. Meyer, A. Murray and K. Reinholtz, *Project Golden Gate: Towards Real-Time Java in Space Missions*, in: *Proceedings of the 7th IEEE International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC), 12-14 May 2004, Vienna, Austria* (2004), pp. 15–22.

[16] Gestegard Robertz, S., R. Henriksson, K. Nilsson, A. Blomdell and I. Tarasov, *Using real-time Java for industrial robot control*, in: *Proceedings of the 5th international workshop on Java technologies for real-time and embedded systems (JTRES)*, 2007, pp. 104–110.

[17] Juillerat, N., S. Müller Arisona and S. Schubiger-Banz, *Real-time, low latency audio processing in java*, in: *Proceedings of the International Computer Music Conference*, Copenhagen, Denmark, 2007.

[18] Kwon, J., A. Wellings and S. King, *Ravenscar-Java: A high integrity profile for real-time Java*, in: *Proceedings of the 2002 joint ACM-ISCOPE conference on Java Grande* (2002), pp. 131–140.

[19] McCloskey, B., D. Bacon, P. Cheng and D. Grove, *Staccato: A parallel and concurrent real-time compacting garbage collector for multiprocessors* (2008).
URL http://www.eecs.berkeley.edu/~billm/rc24504.pdf

[20] Nilsen, K., *Using java for reusable embedded real-time component libraries*, CrossTalk: The Journal of Defense Software Engineering **17** (2004), pp. 13–18.

[21] Puschner, P. and A. Wellings, *A profile for high integrity real-time Java programs*, in: *4th IEEE International Symposium on Object-oriented Real-time distributed Computing (ISORC)*, 2001.
URL http://ieeexplore.ieee.org/iel5/7351/19938/00922813.pdf

[22] RTCA, *Software considerations in airborne systems and equipment certification*, DO-178B, RTCA (1992).

[23] RTCA European Organisation for Civil Aviation Equipment, "ED12B. Software Considerations in Airborne Systems and Equipment Certification," (1992).

[24] Schoeberl, M., *Mission modes for safety critical java*, in: *Software Technologies for Embedded and Ubiquitous Systems, 5th IFIP WG 10.2 International Workshop (SEUS 2007)*, Lecture Notes in Computer Science **4761** (2007), pp. 105–113.
URL http://www.jopdesign.com/doc/scjava_modes.pdf

[25] Schoeberl, M., *Application experiences with a real-time Java processor*, in: *Proceedings of the 17th IFAC World Congress*, Seoul, Korea, 2008.
URL http://www.jopdesign.com/doc/jop_app.pdf

[26] Schoeberl, M., *A Java processor architecture for embedded real-time systems*, Journal of Systems Architecture **54/1–2** (2008), pp. 265–286.
URL http://www.jopdesign.com/doc/rtarch.pdf

[27] Schoeberl, M., H. Sondergaard, B. Thomsen and A. P. Ravn, *A profile for safety critical Java*, in: *10th IEEE International Symposium on Object and Component-Oriented Real-Time Distributed Computing (ISORC'07)* (2007), pp. 94–101.
URL http://www.jopdesign.com/doc/scjava_isorc2007.pdf

[28] Sharp, D. C., *Real-time distributed object computing: Ready for mission-critical embedded system applications*, in: *Proceeding of the 3rd International Symposium on Distributed Objects and Applications, DOA 2001, 17-20 September 2001, Rome, Italy* (2001), pp. 3–4.

[29] Siebert, F., *JEOPARD: Java environment for parallel real-time development*, in: *Proceedings of the 6th International Workshop on Java Technologies for Real-time and Embedded Systems (JTRES 2008)* (2008), pp. 87–93.