A Profile for Safety Critical Java

Martin Schoeberl
Institute of Computer Engineering
Vienna University of Technology, Austria
mschoebe@mail.tuwien.ac.at

Hans Søndergaard
Vitus Bering Denmark University College
DK-8700 Horsens
hso@vitusbering.dk

Bent Thomsen, Anders P. Ravn
Department of Computer Science
Aalborg University DK-9220 Aalborg
bt, apr@cs.aau.dk

Abstract

We propose a new, minimal specification for real-time Java for safety critical applications. The intention is to provide a profile that supports programming of applications that can be validated against safety critical standards such as DO-178B [15]. The proposed profile is in line with the Java specification request JSR-302: Safety Critical Java Technology, which is still under discussion. In contrast to the current direction of the expert group for the JSR-302 we do not subset the rather complex Real-Time Specification for Java (RTSJ). Nevertheless, our profile can be implemented on top of an RTSJ compliant JVM.

1 Introduction

Since the Real-Time Specification for Java (RTSJ) [4] and its more recent revision [11] appeared, there has been much discussion about whether its wide scope, incorporating many dynamic features and many parameters, would allow implementations that would be verifiable such that they can be deployed in high integrity systems. A significant outcome of this discussion is the Ravenscar-Java Profile (RJ) [12], which is based on the Ravenscar profile for Ada [6] and was first published in [14]. RJ is an extended subset of RTSJ that removes features considered unsafe for high integrity systems. Currently another profile for safety critical systems is under discussion in the HIJA forum. What these profiles have in common is a bottom-up approach. Essentially they [4]:

1. take the Java language and its associated virtual machine,
2. provide low level access to physical memory (and interrupts),
3. add an interface to a *scheduler* which is some mechanism that gives predictability to thread execution, and which implements some policy that is specified through *release* and *scheduling* parameters,
4. add an interface to some *memory* areas controlled by mechanisms that may give more predictable allocation of objects,
5. add some mechanism to make synchronization more predictable,
6. add new classes of *asynchronous events* and their *handlers*, and internal event generators called *timers* related to *clocks*,
7. and try to come to terms with *asynchronous transfer of control* and *termination*.

We claim that apart from 1 and 2, the remaining enhancements complicates life for a programmer, because the source program for an application becomes a mixture of application specific requirements (deadlines, periods, binding of external events, and program logic), parameters for controlling policies of the underlying middleware mechanisms (cost, priority, importance, event queue sizes, memory area sizes), and parameters for tuning or sidestepping the mechanisms (miss handlers, timers).

The resulting application programs become inherently complex, and in order to be trustworthy, they must be analyzed by tools before being fielded. It is unclear, whether such tools are part of the profiles or the profiles are intended to be used with standard development tools and validation is seen as an entirely separate process.

The contribution of this paper is a profile that is developed with an entirely different approach. This profile is intended to supply the concepts that support development of small, embedded systems that use the standard patterns advocated by, e.g. the HRT-HOOD methodology [7]. The approach builds on the following:

- the Java language and machine supported by an existing RT profile giving the mechanisms and policies,
- low level access to hardware, since hardware abstraction layers are yet in the future,
- plus *periodic* and *sporadic* threads with application specific parameters, including program logic.

The profile needs support from a compiler enhanced with analysis and synthesis algorithms which for given middleware can compute policy parameters and check their feasibility. Such algorithms may either generate bytecode or be based on macro expansion. The feasibility of the approach is demonstrated by using RTSJ as a middleware.

We start from scratch to build a profile for hard real-time Java for safety critical applications. The aim is:

- An easy to use framework
- Simplified program analysis
- Easy to implement on embedded systems
- Minimum implementation details

A Java programmer with background in real-time systems should be able to use the profile in less than a day.

It is assumed that real-time applications are statically analyzed with respect to worst case execution time, schedulability and memory consumption. The specification shall simplify this essential process.

Section 2 surveys the features of existing related profiles and discusses the distinctions we make between mechanisms, policies and application relevant concepts. According to our design principles, we remove the former two from the API. Section 3 defines the resulting profile and illustrates it with small examples. Section 4 outlines an implementation. Section 5 concludes the paper with a discussion of some features that might be nice to have or which may be supported in other ways.

2 Selection of Features

In the following we discuss each of the areas where RTSJ extends Java, and where Ravenscar-Java and Safety Critical Java decide to restrict it. The discussion follows the structure of the RTSJ [4].

2.1 Standard Java Classes

As specified in RTSJ, some features of standard Java classes dealing with priorities, thread groups, interrupted exception and system properties have to be redefined, to the extent that they remain in CDLC [21]. We agree with RTSJ on these points.

2.2 Threads

A *RealtimeThread* is a fundamental concept; but in its constructor we see the complexity of the policies and mechanisms of middleware. In RTSJ it is parameterized by: Scheduling Parameters, Release Parameters, Memory Parameters, Memory Area, Processing Group, and finally a Run Logic. According to our principles, we keep the application specific Release Parameters and Run Logic, while the mechanism and policy dependent parameters are elided.

The concept of a *NoHeapRealtimeThread* is a restriction in both Ravenscar-Java and in Safety Critical Java. It is clearly an attempt at supporting static analysis by prohibiting general dynamic memory allocation. Our viewpoint is that analysis of memory usage is better left to a tool, thus the concept disappears. In the underlying platform there might be a real-time garbage collection with guaranteed performance, and that should not be a priori eliminated as an implementation option.

Ravenscar-Java introduced an *Initializer* thread. However, since the assumption is that the system is initialized once only, this might as well be done through the usual *main* method of an ordinary static *System* class. The definition of an Initialization thread that has the highest priority to perform all the initialization is an implementation artefact[1] and should not be part of the specification. It makes it unnecessary troublesome to find a place for implementation code that has to run just before the mission phase (e.g., generation of the PianoRoll on the aJile processor [20]).

The Run Logic should in both cases be a single method, and the cleanest way of introducing them in an object-oriented style, when they are fixed for the classes, is through specialization of an abstract run method in the threads. The waitForNextPeriod is eliminated, because it is redundant. The logic should not be subdivided into phases (by waitForNextPeriod), because it would hamper WCET analysis.

During the mission phase, an application has a fixed number of *sporadic* or *periodic* threads. We thought quite a lot about having several modes and the possibility of being able to re-initialize the system when changing from one mode to another, but decided against it in the core definition. The parameters for these threads are discussed below.

[1] It is a result from RJ implementation experiments on top of RTSJ. However, this *trick* can be used, but as part of the implementation.

2.3 Release Parameters

On an RTOS the *importance* of a task is usually assigned by a priority. However, in real-time systems the only relevant parameters are release times. A mapping of timing requirements to priority may be done by rate-monotonic assignment [13] or more general by deadline-monotonic assignment [2] or any other policy that is compatible with the scheduler of the middleware.

In the proposed specification the release parameters of schedulable entities (periodic and sporadic threads) are time and not priority. An implementation that uses a priority based preemptive scheduler may map the time requirements according to the deadline-monotonic order. As we do not allow dynamic creation of threads during mission phase this mapping can easily be done on the transition from the initialization to the mission phase. Furthermore, it allows, e.g. EDF scheduling in the middleware without changing the application.

Common for application threads is a *deadline*, given as a *RelativeTime*. For periodic threads, the *period* is also application specific and it is a relative time; similarly for sporadic threads, we have a *minInterarrivalTime*.

Cost is a derived property, the WCET of the run logic. It is not a quantity that application programmers should code up in a safety critical system. Either it can be computed, i.e. checked by a tool or the logic needs to be simplified. Thus we do not want *cost* as part of the profile. Absolute time is usually not available in small systems, and absolute *start* times seem hard to validate offline – when are they invalid? The various miss- and overrun-*handlers* should not occur, cf. Ravenscar-Java or the SC proposal. The most one can do seems to be a static emergency stop that brings the application to a safe state – hopefully.

2.4 Memory Management

The very detailed classification of memory and its handling reminds at least one of the authors of the sophisticated language features that in the middle of last century were added to languages like PL/I and COBOL to allow programmers to specify *overlays*. They were probably never used, because virtual memory very soon provided adequate solutions. We boldly suggest that such analysis is better done by tools, such as region analysis for functional languages [22], adapted for real-time java in [5] or by Scoped Types recently proposed in [23] and further elaborated in [1].

RawMemoryAccess remains unavoidable, as long as no standardized hardware abstraction layers exist. It is a very unsafe feature as evidenced by the many efforts to analyze driver code for anomalies in standard operating systems. Perhaps it should be restricted to library modules for most applications.

2.5 Synchronization

In order to keep scheduling predictable, synchronization around protected regions must implement a suitable protocol. We see this as a product of the analysis and synthesis, not as a task for the programmer.

2.6 Asynchronous Events and their Handlers

Software interrupts is a programming paradigm that makes systems very hard to analyze, because they entail some sort of buffering mechanism; thus we suggest that Asynchronous Events are restricted along with their handlers. The remaining use for events is to connect sporadic threads to the external interrupts that they handle. We propose that this is done directly during initialization through the standard uninterpreted *happening* token.

2.7 Remaining Details and Summary

The profile is summarized below and a prototype implementation has been developed to support it. However, it would be misleading to claim that it is fully worked out, because for high integrity systems, the code must also be checked for potential uncaught exceptions. This is not part of the syntactic extensions; but a matter for analysis tools.

3 The SCJ Proposal

This section describes our *Safety Critical Java Profile* proposal.

3.1 Overview of the SC Java Profile

As discussed in Section 2, the profile should be small and simple to use when modeling an application, but still at least as powerful as the Ravenscar-Java profile. In the javax.safetycritical package you first and foremost find the thread classes for the periodic time-triggered activities, and for the event-triggered sporadic activities.

The RealtimeSystem class is the representation of the real-time runtime system. Time is represented by the RelativeTime class, and we also have classes for access to physical memory. They are a very simpleminded hardware abstraction layer.

The rest is hidden for the programmer or included in analysis tools that are specific for given middleware policies and mechanisms, so that the platform independence of the application is fulfilled.

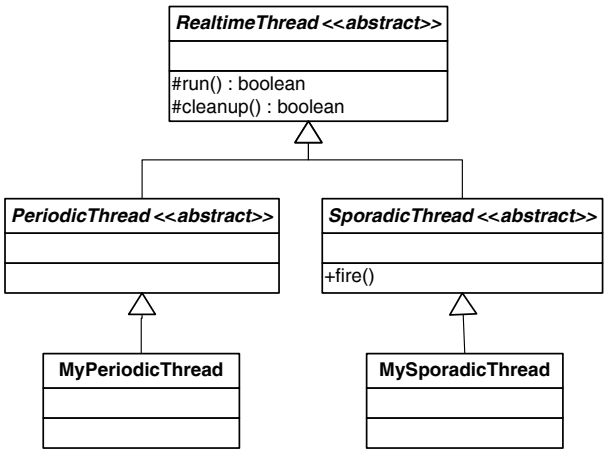

Figure 1. Class structure of the profile with application threads

3.2 Schedulable Entities

All schedulable entities (periodic and sporadic) are represented by threads. The structure of real-time threads is shown in Figure 1. The abstract class RealtimeThread has two methods:

- run() the run logic to be executed every time the thread is activated
- cleanup() a clean-up method to be executed if the system should be shut down or stopped

We have considered two different design patterns for specializing it: through *subclassing* or through *delegation* to a Runnable via a parameter in the constructor. If subclassing is used, the run method with the run logic is implemented in a subclass of PeriodicThread or SporadicThread classes. This supports the static architectural patterns that we envisage for applications. In contrast, delegation through a parameter gives an illusion that the run logic might be changed dynamically during execution. As a result of the arguments in Section 2.2, we have in this profile chosen to introduce the run logic through subclassing. This also means that the PeriodicThread and SporadicThread classes are declared abstract, see Figure 1.

For sporadic events we bind each event to a single thread. The event/thread relation is a 1:1 relation (same as in the original Ravenscar-Java, different to the RTSJ). Therefore, periodic threads and sporadic event threads are very similar. Both contain program logic that gets released by an *event*. For a periodic thread this event is generated by the elapsing time, for a sporadic thread either by an hardware event (interrupt) or an software event (invocation of fire()). Figure 1 shows the simple class hierarchy.

```
package javax.safetycritical;

public abstract class RealtimeThread {

    protected RealtimeThread(RelativeTime period,
        RelativeTime deadline,
        RelativeTime offset, int memSize)

    protected RealtimeThread(String event,
        RelativeTime minInterval,
        RelativeTime deadline, int memSize)

    abstract protected boolean run();

    protected boolean cleanup() {
        return true;
    }
}
```

Figure 2. The base class for all schedulable entities

We use a single class (RealtimeThread) to express all schedulable entities (periodic time-triggered, software event-triggered, and hardware event-triggered). Figure 2 shows the definition of RealtimeThread.

The run() method has to be overwritten for the application logic – that is the same abstraction as in standard java.lang.Thread. The difference to j.l.Thread is that the method is abstract and it has to return a boolean result. Declaring it abstract forces the programmer to implement run(). The return value is used for the shutdown as described in Section 3.3.1. The method cleanup() is invoked during the shutdown phase.

For a periodic real-time entity the run() method is periodically invoked, for a sporadic entity when the software or hardware event happens. We are not using the loop construct with waitForNextPeriod() as in the RTSJ.

The class is abstract and extended by PeriodicThread and SporadicThread. Figure 3 shows the two class definitions. PeriodicThread is just a wrapper for a periodic RealtimeThread. SporadicThread has an additional method fire() to implement software events. After invoking fire() on a SporadicThread, the thread gets released by the scheduler.

3.2.1 Release Parameters

All time values are specified by the class RelativeTime which is similar to the same class in the RTSJ. However, we remove all methods that can change the time value. The resulting objects are immutable. Immutable time values guarantee that the release parameters cannot be changed during the mission phase.

```
package javax.safetycritical;

public abstract class PeriodicThread
        extends RealtimeThread {

    public PeriodicThread(RelativeTime period,
        RelativeTime deadline,
        RelativeTime offset, int memSize)

    public PeriodicThread(RelativeTime period)
}

public abstract class SporadicThread
        extends RealtimeThread {

    public SporadicThread(String event,
        RelativeTime minInterval,
        RelativeTime deadline, int memSize)

    public SporadicThread(String event,
        RelativeTime minInterarrival)

    public void fire()
}
```

Figure 3. The wrapper classes for periodic and sporadic real-time events

The parameters period and deadline represent the period and the deadline for periodic threads. The start time relative to the mission start is given with offset.

Parameter event (the *happening* token) describes the source for a hardware event (e.g., "INT1" for an interrupt handler of interrupt 1). The value of the event string is of course system specific. Parameter minInterarrival is the allowed minimum inter-arrival time for an event. For a software event, the runtime system can enforce this limit. However, for hardware generated event this is usually not possible on a common microcontroller. We can restrict the release of the sporadic thread, but not the invocation of the scheduler. Therefore, an interrupt burst can disturb the real-time schedule. We plan to add a special interrupt controller to JOP [17] to enforce this minimum inter-arrival time at the hardware level.

Note that these time quantities correspond to natural requirements of an application. They are not parameters for tuning or controlling scheduling mechanisms.

3.2.2 Memory Model

Each thread has an associated scoped memory for dynamic data. The parameter memSize in the constructor determines the size of the scoped memory for a thread. This scoped memory is similar to the scoped memory in the RTSJ, but it cannot be shared between threads. It is entered before the invocation of run() and exited after the return from run(). The whole scoped memory is cleared on exit. That means that all memory allocation requests can be performed in constant time.

However, this single scope for a thread and the size as a parameter for the thread constructor is an intermediate solution. In future work we will provide compiler generated scopes.

```
package javax.safetycritical;

public class RealtimeSystem {

    private RealtimeSystem()

    public static void start()
    public static void stop()
    public boolean shutdownPending()
    public static int currentTimeMicros()
}
```

Figure 4. The representation of the real-time system

3.3 Initialization, Mission, and Stop

Figure 4 shows the class definition of RealtimeSystem, our representation of the real-time runtime system. RealtimeSystem is similar to java.lang.System and cannot be instantiated. Using only static methods is the simplest realization of a singleton.

During the lifetime of an RT Java application, it will be in one of the three distinct states (as shown in Figure 5), with well-defined rules that define the transitions between the states:

Initialized: An RT application is in the *Initialized* state until the initialization code of the RealTimeSystem has run to completion and its start method has not been invoked. Application threads and passive objects are created and initialized here. Threads are not started.

Mission: An RT application is in the *Mission* state when it has returned from its start method which starts all threads.

Stop: An RT application is in the *Stop* state when it has returned from the stop method which waits for threads to perform their optional cleanup.

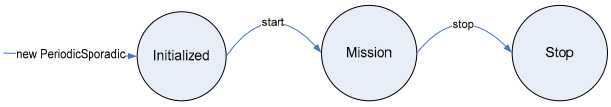

Figure 5. Application states

Those three states are inspired by the states in a MIDlet, but they are also based on industrial experiences from real-time systems which need to shut down in a controlled way.

3.3.1 Shutdown

In contrast to RJ we provide an additional phase for the real-time application: *Shutdown*. This phase is intended to provide a safe termination of the real-time system; e. g. threads have a chance to bring actuators into a safe position. The shutdown phase is initiated similar to the start of the mission phase, by invoking stop() from RealtimeSystem.

However, we can not simply stop all threads, but need a form of cooperation. All real-time threads return a boolean value from the run() method. This value indicates: *I'm ready for a shutdown*. When a thread is in a critical operation, where a shutdown is not allowed, the thread just return false to delay the shutdown process. In the first sub-phase the runtime system waits for *all* threads to be ready for shutdown before actually switching to *cleanup*.

In the second sub-phase, all tasks are notified through calls to the cleanup() method that clean up is in progress. During the shutdown the cleanup() method is invoked periodically[2] *instead* of the run() method. The cleanup method returns true to signal that its task has safely completed its shutdown responsibilities. If the return value is false, cleanup() will be called again; if the return value is true, the thread is terminated. The possibility of several cleanup invocations allow threads to resolve mutual dependencies during shutdown and spread the cleanup work so that all tasks can still meet their deadlines. Shutdown terminates when all threads are terminated.

3.4 An Example

Figure 6 shows a short example how to write a real-time application with the proposed profile. One periodic thread prints a message each second and fires the software event on every second iteration. The sporadic thread handles the software event and prints a message. After 10 iterations the periodic thread requests a shutdown from the runtime systems. On behalf of this shutdown the method cleanup() gets invoked once.

[2] at the same period as the former run() method

```
public class PeriodicSporadic {

   public static void main(String[] args) {

      final SporadicThread rte =
         new SporadicThread("SWEVENT",
            new RelativeTime(2000, 0)) {

         protected boolean run() {
            System.out.println("SW event fired");
            return true;
         }
      };

      new PeriodicThread(
        new RelativeTime(1000, 0)) {
         int counter = 0;
         protected boolean run() {
            System.out.println("P1");
            ++counter;
            if (counter%2==1)
               rte.fire();
            if (counter==10)
               RealtimeSystem.stop();
            return true;
         }

         protected boolean cleanup() {
            System.out.println("cleanup!");
            return true;
         }
      };

      RealtimeSystem.start();
   }
}
```

Figure 6. An example of a real-time application

4 Implementation

It is mandatory to provide a reference implementation for the specification and evaluate it with some application test cases. An implementation of a first draft of the proposed specification is available on the Java processor JOP [17], as well as on the aJ100. The implementations are light-weight and use the already available real-time thread implementations [16, 20]. In [16] it has been shown that the JOP scheduler performs quite will. Tasks with periods down to 100 μs can be scheduled on a 100 MHz version of JOP. The scheduling code base is industry proven as it is in production in three applications.

Furthermore, we are currently also working on an imple-

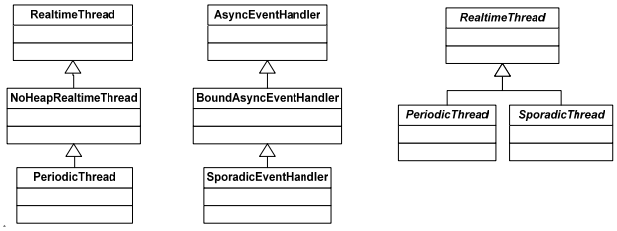

Figure 7. Thread and Handler classes: RJ (left), our SCJ proposal (right)

mentation on top of the RTSJ [4]. We use the Sun version (Mackinac [3]) on top of Solaris for a standard PC. So far we have not found any issues that make it impossible to implement the proposed specification on top of the RTSJ.

4.1 Structuring the API

To obtain compatibility with the RTSJ a safety critical API can be structured in at least three ways:

- as a subset of RTSJ at the method level
- as a new API with an RTSJ-compliant binding
- mirroring the RTSJ class structure

Ravenscar-Java has chosen the first approach. We have chosen the second approach in our profile.

The two different approaches mirror two of the most common techniques for reusing functionality in object-oriented systems [9]: *class inheritance*, also called *white-box* reuse and *object composition*, also called *black-box* reuse.

At the definition of real-time threads in RTSJ (and consequently in Ravenscar-Java) the class inheritance pattern has been used. This approach has shown to be a problematic design choice, resulting in inheritance of all the public methods from the java.lang.Thread class. In contrast to this, using the object composition pattern results in small and clean thread classes with precisely those methods you want to specify.

Asynchronous events and their handlers were in RTSJ introduced almost totally separated from the real-time threads. However, over time a better classification of events in periodic events (time-triggered) and sporadic events (event-triggered) has evolved. Yet, structuring a profile as a subset of RTSJ using the class inheritance pattern complicates the design of a new profile, see Figure 7. Also here the object composition pattern gives a cleaner solution.

4.2 Verification and Tools

A real-time application has to be verified, at minimum the WCET and a schedulability analysis. For usage of dynamic memory we propose automatic/compiler generation of scoped memory. The worst-case memory consumption (WCMC) has to be analyzed.

For the implementation on JOP we can use the available WCET analyzer [18]. However, a tighter integration with the profile (e.g., automatic selection of the run() method to be analyzed) and a schedulability analyzer is still missing.

4.3 Standard Library

The RTSJ assumes the standard JDK or at minimum a J2ME verison of the JDK (CLDC or CDC). However, the specification is very silent about how and if those library functions can be used within a real-time application. We need following information for the library:

1. Dynamic memory allocation
2. Worst-case execution time
3. Blocking time

There is not much work available how to solve those issues. The Javolution project [8] provides substitutes for the collection classes where elements are recycled. In that case we do not need a garbage collector for collections.

In [10] the Java Modeling Language (JML) is used to describe pre- and post-conditions for methods to describe loop bounds. The descriptions are than proven by KeY, a semi-automated prover. Data flow analysis propagates the bound information to all call sites and the results can be integrated into an ILP based WCET analyzer (e.g., [18]).

Exceptions need dynamic memory and are an issue e.g., for the implementation of an RTSJ compliant JVM. A pragmatic, but not correct, approach is to pre-allocate the data structures for all possible exceptions in immortal memory and reuse them. Besides need for dynamic memory exceptions are also problematic with respect to WCET. A better approach is to avoid exceptions by a defensive programming style. The absence of runtime exceptions can then be proved formally [19].

5 Conclusion

In this paper we have presented a simple real-time Java profile for safety critical systems. Instead of building on top of RTSJ using the class inheritance pattern, we have designed our profile by using the object composition pattern.

Simultaneously, we have gone through RTSJ, Ravenscar-Java, and the SC Java proposal to find the points where these

become very policy and mechanism dependent, hereby separating the features for the profiles in application specific requirements (periods, etc.), policies for the middleware mechanisms (priorities, etc.), and tuning mechanisms (miss handlers, etc.).

By this we have been able to specify a simple safety critical Java profile with very clean classes which only include the application specific requirements. The check whether the specified application is feasible for a platform with specific middleware should be statically analyzed by appropriate analysis tools.

Our profile will without problems be able to run on top of existing RTSJ or Ravenscar-Java implementations. The next steps will be to specify and implement the profile in all details; especially development of the tools which are part of the profile. Furthermore, we will evaluate different provisions for application mode changes as an extension to the profile.

Acknowledgment

The first author would like to thank Franz Josef Jappel for discussions on a usable API for real-time systems and various suggestions.

References

[1] C. Andreae, Y. Coady, C. Gibbs, J. Nobble, J. Vitek, and T. Zhao. Scoped types and aspects for real-time java. In *Proceedings of the European Conference on Object Oriented Programming (ECOOP'9)*, volume 4067 of *LNCS*, pages 124–147. Springer-Verlag, July 2006.

[2] N. C. Audsley, A. Burns, M. F. Richardson, and A. J. Wellings. Hard real-time scheduling: The deadline monotonic approach. In *Proceedings 8th IEEE Workshop on Real-Time Operating Systems and Software*, Atalanta, 1991.

[3] G. Bollella, B. Delsart, R. Guider, C. Lizzi, and F. Parain. Mackinac: Making HotSpot^{TM} real-time. In *ISORC*, pages 45–54. IEEE Computer Society, 2005.

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

[5] C. Boyapati, A. Salcianu, J. William Beebee, and M. Rinard. Ownership types for safe region-based memory management in real-time java. In *PLDI '03: Proceedings of the ACM SIGPLAN 2003 conference on Programming language design and implementation*, pages 324–337, New York, NY, USA, 2003. ACM Press.

[6] A. Burns, B. Dobbing, and G. Romanski. The ravenscar tasking profile for high integrity real-time programs. In *Proceedings of the 1998 Ada-Europe International Conference on Reliable Software Technologies*, pages 263–275. Springer-Verlag, 1998.

[7] A. Burns and A. J. Wellings. Hrt-hood: a structured design method for hard real-time systems. *Real-Time Syst.*, 6(1):73–114, 1994.

[8] J.-M. Dautelle. Validating java for safety-critical applications. In *AIAA Space 2005 Conference*, 2005.

[9] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns*. Addison Wesley Professional Computing Series. Addison-Wesley, 1995.

[10] J. J. Hunt, F. B. Siebert, P. H. Schmitt, and I. Tonin. Provably correct loops bounds for realtime java programs. In *Proceedings of the 4th international workshop on Java technologies for real-time and embedded systems (JTRES 2006)*, pages 162–169, New York, NY, USA, 2006. ACM Press.

[11] Java Expert Group. Java specification request 282: RTSJ version 1.1, September 2005.

[12] J. Kwon, 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*, pages 131–140. ACM Press, 2002.

[13] C. L. Liu and J. W. Layland. Scheduling algorithms for multiprogramming in a hard-real-time environment. *J. ACM*, 20(1):46–61, 1973.

[14] P. Puschner and A. J. Wellings. A profile for high integrity real-time Java programs. In *4th IEEE International Symposium on Object-oriented Real-time distributed Computing (ISORC)*, 2001.

[15] RTCA/DO-178B. Software considerations in airborne systems and equipment certification. December 1992.

[16] M. Schoeberl. Restrictions of Java for embedded real-time systems. In *Proceedings of the 7th IEEE International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC 2004)*, pages 93–100, Vienna, Austria, May 2004.

[17] M. Schoeberl. *JOP: A Java Optimized Processor for Embedded Real-Time Systems*. PhD thesis, Vienna University of Technology, 2005.

[18] M. Schoeberl and R. Pedersen. WCET analysis for a Java processor. In *Proceedings of the 4th international workshop on Java technologies for real-time and embedded systems (JTRES 2006)*, pages 202–211, New York, NY, USA, 2006. ACM Press.

[19] F. Siebert. Proving the absence of RTSJ related runtime errors through data flow analysis. In *Proceedings of the 4th international workshop on Java technologies for real-time and embedded systems (JTRES 2006)*, pages 152–161, New York, NY, USA, 2006. ACM Press.

[20] H. Sondergaard, B. Thomsen, and A. P. Ravn. A ravenscar-java profile implementation. In *Proceedings of the 4th international workshop on Java technologies for real-time and embedded systems (JTRES 2006)*, pages 38–47, Paris, France, October 2006. ACM Press.

[21] Sun. Java 2 platform, micro edition (J2ME). Available at: http://java.sun.com/j2me/docs/.

[22] M. Tofte, L. Birkedal, M. Elsman, and N. Hallenberg. A retrospective on region-based memory management. *Higher-Order and Symbolic Computation*, 17(3):245–265, 2004.

[23] T. Zhao, J. Noble, and J. Vitek. Scoped types for real-time Java. In *Proceedings of the 25th IEEE International Real-Time Systems Symposium (RTSS'04)*, pages 241–251, Washington, DC, USA, 2004. IEEE Computer Society.