Certifiable Java for Embedded Systems

Martin Schoeberl
Technical University of Denmark
masca@dtu.dk

Andreas Engelbredt Dalsgaard, René Rydhof Hansen
Dept. of Computer Science
Aalborg University
{andrease, rrh}@cs.aau.d

Stephan E. Korsholm
VIA University College, Horsens
sek@via.dk

Anders P. Ravn
Dept. of Computer Science
Aalborg University
apr@cs.aau.dk

Juan Ricardo Rios Rivas, Tórur Biskopstø Strøm
Technical University of Denmark
jrri@dtu.dk,
torur.strom@gmail.com

Hans Søndergaard
VIA University College, Horsens
hso@viauc.dk

ABSTRACT

The Certifiable Java for Embedded Systems (CJ4ES) project aimed to develop a prototype development environment and platform for safety-critical software for embedded applications. There are three core constituents: A profile of the Java programming language that is tailored for safety-critical applications, a predictable Java processor built with FPGA technology, and an Eclipse based application development environment that binds the profile and the platform together and provides analyses that help to provide evidence that can be used as part of a safety case. This paper summarizes key contributions within these areas during the three-year project period. In the conclusion the overall result of the project is assessed.

Categories and Subject Descriptors

D.4.7 [**Operating Systems**]: Organization and Design—*Real-time systems and embedded systems*

Keywords

Safety-critical Java

1. INTRODUCTION

Embedded systems are increasingly becoming part of our daily life. Some of these systems, for example the control of the Copenhagen Metro, are safety-critical, as our *real* life can depend on it. Such systems need to be certified to be used safely.

There are an increasing number of such systems in the world and even more are planned. Therefore, a restricted version of Java [10] is considered as a new platform to develop certifiable software for future safety-critical and real-time applications. This subset is called Safety-Critical Java and specified under the Java Community Process as Java Specification Request (JSR) 302 [17].

However, a Java profile, although an important part, is not enough to build trustworthy systems. It has to have a dependable platform, and a development environment that binds the profile and the platform together and provides analyses that help to provide evidence that can be used as part of a safety case.

The objective of the Certifiable Java for Embedded Systems (CJ4ES) project was to investigate these areas and in this way consolidate and integrate a number of results from previous research by members of the team. Key previous results include development of a time-predictable Java processor [34, 36], contributions to profiles for predictable Java [5, 10, 43], and development of analysis tools [6, 41]. As most of the previous results are open-source,[1] the results from this project are available as open-source as well.

The safety-critical Java (SCJ) standard needs to be evaluated by implementing it and using it in example applications. Within the CJ4ES project we have explored the standard, provided two implementations of it, built example applications, investigated the programability of SCJ, and last but not least, given valuable feedback to the expert group of JSR 302.

This paper summarizes and reflects on the work and is organized in 11 sections. The following section presents related work and Section 3 provides background on safety-critical Java. Section 4 presents two JVMs, the Java processor JOP and the hardware near VM (HVM), used as platforms for our SCJ implementations, while Section 5 describes the two SCJ implementations. Section 6 describes two tools: the memory safety analysis tool and the worst-case memory consumption analysis tool. Section 7 describes the testing framework for SCJ. Section 8 presents hardware support for SCJ within JOP. Section 9 and Section 10 explore the expressiveness of SCJ with implementations of libraries and applications. Section 11 concludes the paper.

[1]see `https://github.com/jop-devel/jop` and `http://www.icelab.dk/download.html`

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from Permissions@acm.org.
JTRES'14, October 13–14 2014, Niagara Falls, NY, USA
Copyright 2014 ACM 978-1-4503-2813-5/14/10 ...$15.00.
http://dx.doi.org/10.1145/2661020.2661025.

2. RELATED WORK

Java for real-time systems started with work on PERC Pico [22]. Later on, the Real-Time Specification for Java (RTSJ) [7] was defined. RTSJ initiated the Java Community Process and is therefore the first Java specification request (JSR 1). However, RTSJ is considered too complex to build safety-critical systems. Therefore, work started to simplify RTSJ.

The SCJ specification is based on early attempts to simplify the RTSJ for high integrity applications. Puschner and Wellings presented the first proposal of a subset of the RTSJ classes [26]. They introduced the concept of an *initialization* and *mission* phase into real-time Java. All threads, event handlers, and shared objects are set up in the initialization phase. The restrictions (e.g., static priorities, no call of sleep, no `wait/notify`, and no dynamic class loading) are very similar to the restrictions of the level 1 of SCJ.

The work was refined and renamed to Ravenscar Java [15] to emphasize the heritage of the concepts from the Ada Ravenscar tasking profile [8]. Later proposals for a safety-critical Java profile argue for an API that is independent of the RTSJ [33, 43]. The argument is that all approaches that inherit from RTSJ classes introduce additional complexity. Furthermore, two versions of the RTSJ classes (the original and the restricted) may confuse real-time Java programmers. The issue comes from the fact that the RTSJ is more expressive than SCJ, but SCJ classes extend the RTSJ classes. Therefore, the RTSJ classes in the SCJ version need to be restricted. To model this *inverse* relation between RTSJ and SCJ it has been proposed to build the class hierarchy the other way round: RTSJ shall inherit from classes as defined in SCJ [5].

Søndergaard et al. provide an implementation of the Ravenscar Java profile [46]. The implementation targets industrial applications and uses an aJ-100 processor developed by aJile Systems [1]. The aJ-100 is a 32-bit microprocessor that directly executes JVM bytecodes (implemented in microcode) as its native instruction set. In addition, it provides a microcode programed real-time kernel that provides, among others, support for scheduling, context switching and object synchronization.

Plsek et al. present one of the first implementations of SCJ on an embedded platform [24]. They provide an implementation of SCJ's Level 0 running on the OVM virtual machine [4]. The OVM is a framework that enables alternate implementations of core VM functionality (e.g., different versions of priority inheritance monitors) in order to build and test VMs with different features. OVM uses an ahead-of-time compiler to translate Java code to C++ and then it uses the GCC compiler to obtain machine code. SCJ's implementation on OVM runs on an FPGA board executing the RTEMS real-time operating system on a LEON3 processor.

In the work presented by Bøgholm et al. in [5] a different approach is taken. Instead of using the classes defined by the SCJ profile, their implementation is based on a profile called Predictable Java (PJ). PJ is a Java profile suitable for the development of high-integrity real-time embedded systems that builds on the ideas of [15, 26, 43] and [46]. The profile is based on the execution of event handlers grouped in missions, which in turn are also considered event handlers. Furthermore, the profile considers that, as PJ classes are more restricted in functionality than RTSJ classes, PJ should be a generalization and not a specialization of RTSJ. In contrast, SCJ classes as defined by JSR-302 are a specialization of RTSJ classes.

The upcoming version of PERC Pico [3] will be SCJ compliant. However, Atego considers implementing SCJ in addition to the current PERC Pico notion of safety-critical Java. The intention is to support both APIs and memory models in a single JVM [21]. That paper also describes the differences between the SCJ memory model and the PERC Pico memory model.

3. SAFETY CRITICAL JAVA

Safety-critical Java (SCJ) [17] was developed, within the Java Community Process (JCP) as Java specification request number JSR 302. SCJ shall be a high-level platform for future safety-critical real-time systems, with special focus on systems that require formal certification. In order to achieve this, the SCJ is defined as a restricted subset of the Real-Time Specification for Java (RTSJ) [7], with additional classes defined. It is defined in such a way that it can be implemented and run directly on top of RTSJ, which is indeed how the SCJ reference implementation is made. However, for certification purposes this is not recommended since that would also require certification of the full RTSJ.

To facilitate the use of SCJ across a wide spectrum of applications covering a great variety of safety requirements, three *levels* are defined by the SCJ, with increasing complexity and expressive power for the application programmer, at the cost of increased effort needed for safety certification. At the most restrictive level, termed *level 0*, only single threaded applications running under a cyclic executive are supported.

A level 0 application can use all the memory areas defined by the SCJ: immortal, mission, and private (see below for more details on the SCJ memory areas). Since every handler instance is allowed to run to completion, i.e., it is not preempted, the *backing store* for a handler's private memory can be reused by other handler instances. Level 1 introduces preemptive scheduling with ceiling based locks. Furthermore, interrupt handlers, written in Java, are allowed in level 1. In level 2 *nested missions* are introduced, allowing for more dynamic systems to be implemented in SCJ, in which It is possible to keep part of the system running and starting and stopping other parts during runtime. Level 2 also introduces an adapted version of RTSJ's `NoHeapRealtimeThread`.

For all three levels, the specific size of the needed backing store must be explicitly given by the application programmer as part of the SCJ program. Taking the complexity of an SCJ implementation and the underlying (hardware) platform into account, it is a non-trivial task to correctly estimate the (worst-case) memory consumption of a given SCJ application. In Section 6.2 a tool for automating this task is discussed. The SCJ memory model is discussed in more detail in Section 3.2.

Managed schedulable objects (MSO) represent concurrent activities in SCJ. MSOs are objects that are managed by a mission and scheduled for execution by the system's scheduler. In level 1 applications, MSOs are called *handlers*, similar to RTSJ-style event handlers. In fact the SCJ handlers are a subclass of RTSJ's `BoundAsyncEventHandler`. Handlers come in two flavors: (1) periodic event handlers (PEH) that are released in a time-triggered fashion and (2) aperiodic handlers (AEH) that are released by specific events. The event to release an AEH can be a software event or an interrupt.

3.1 Missions and Scheduling

One or more *missions* that are executed in a sequence controlled by a *mission sequencer* comprise an SCJ application. Missions can be used to represent different phases or operational modes of an application. The mission itself consists of a set of MSOs along with its mission memory.

A mission has three phases: initialization, execution, and cleanup. In the initialization phase the mission memory is created by the SCJ implementation, all MSOs belonging to the mission (during the entire lifetime of the mission) are created, and data created during initialization is by default allocated in the mission memory. Data shared between handlers must be allocated in mission memory, while data shared between missions must be allocated in immortal memory.[2]

The application itself is started upon transitioning from the initialization phase to the execution phase. Once the application is executing, no new MSOs can be registered or started and temporary objects are allocated in the MSO's private memory. While it is still possible to allocate space in mission memory, it is strongly discouraged. After the cleanup phase, the mission memory is cleared and a new mission can be started.

In order to implement an SCJ application, an implementation of the `Safelet` interface must be provided and, in addition, at least one class that extends the `Mission` class. Simple applications, consisting of a single mission, can use one class that extends `Mission` and implements `Safelet`.

3.2 The Memory Model

In an effort to make the execution time behavior of an application more predictable, the SCJ memory model does *not* include a traditional heap with a garbage collector. As hinted at in the preceding sections, SCJ instead uses a memory model based on *scoped memory areas*. These are memory areas that are explicitly, and thus manually, managed by the application programmer and thus free of garbage collector interference. The SCJ memory model defines three different scoped memory areas: immortal memory, mission memory, and private memories. Immortal memory contains the static fields, objects created during class initialization, and application data that may be needed throughout the application's lifetime. This is similar to the RTSJ notion of immortal memory. Unsurprisingly, mission memory only exists for the duration of the mission (in any of the three mission phases). Finally, private memories are used by a single MSO and cannot contain shared objects.

While private memory areas in the SCJ memory model may be nested, they cannot be shared between different threads of execution. Consequently, the memory scopes that are used by an MSO, at any point during execution, are structured as a simple *stack* of scopes. In this scope stack recently allocated and shorter-lived memory areas can be found near the top and longer-lived memory areas are conversely located nearer the bottom of the stack. This is in contrast to the scoped memory model of the RTSJ, where memory areas may be nested and shared in complex patterns, leading to a so-called "cactus stack" structure for memory scopes.

With the introduction of programmer managed memory areas, and thus the lack of garbage collection, also comes the risk of *dangling references*. A dangling reference is the result of (unintentionally) deallocating a memory area that is still in use, i.e., there are still references pointing inside the memory area. In order to avoid dangling references, the SCJ memory model dictates that reference assignments are allowed only when the reference points to an object stored in a memory area at a *deeper* level in the scope stack, i.e., references from the current memory are only allowed to point to objects that live in a memory area with a longer life-time than the current one. For an application programmer, it can be very difficult to keep track of the current scope stack and thus may inadvertently introduce an illegal reference to a shorter-lived memory area. In Section 6.1 we describe how an automated tool can be used by programmers to verify that a program does not contain any (potentially) illegal references.

4. JAVA VIRTUAL MACHINES FOR SCJ

Within the CJ4ES project we developed and used two different Java virtual machines: (1) the JOP processor [36] as a hardware implementation in an FPGA, and (2) the hardware near virtual machine [14] as a software VM targeting small embedded processors with small memory constraints.

Both JVMs share ideas of how a hardware abstraction layer in Java can be implemented [39]. Access to IO devices is supported by so called *hardware objects*, where object fields are mapped to device registers. Larger, memory mapped IO areas can be mapped to Java arrays. The other interaction with IO devices, interrupts, is supported by first level interrupt handlers in Java, where the interrupt handler object is used for synchronization between Java threads and the interrupt handler. The interrupt handlers support the notion of hardware priorities, which are above software priorities.

4.1 The Java Optimized Processor

The Java optimized processor (JOP) [36] is a hardware implementation of the JVM, a Java processor. The JVM bytecodes are mapped to a fixed sequence of microcode instructions. The main benefit of this hardware implementation of the JVM is the well-known and mostly constant execution time of individual bytecodes. JOP also contains three special forms of caches: (1) a method cache to cache whole methods, (2) a stack cache for stack allocated data, and (3) on object cache for heap allocated objects.

All three caches are intended to lower the worst-case execution time (WCET) and simplify the WCET analysis. The method cache loads methods only on method invocation or on a method return. Therefore, all other bytecodes are guaranteed hits in the cache. WCET analysis needs only consider the call tree.

In the JVM the stack is used for return information, local variables, and as operand stack. Almost all bytecodes access data allocated on the stack. Therefore, the stack area needs caching. Stack addresses are relative simple to track and therefore a stack cache is easy to analyze. In the current version of JOP this is even further simplified as all stack allocated data of a task need to fit into the stack cache. Therefore, it can be ignored by WCET analysis, as all accesses are hits by design. The stack content is exchanged

[2]For a level 2 application, data shared between missions can also be allocated in the mission memory of an enclosing mission.

with the main memory at task switch only.

Addresses of heap allocated data are known only at runtime. Therefore, address based cache analysis fails for this type of data. With JOP's object cache [38] objects are tracked symbolically with a high associativity. This enables WCET analysis even of heap-allocated data [12].

The time-predictability of bytecode execution and the special caches enable static analysis of the WECT of Java programs [41]. Therefore, JOP is a good platform for safety-critical applications where the WCET needs to be statically determined.

JOP is available as single core or a multicore version. The multicore version includes time-predictable memory arbitration [23]. Therefore, even on a multicore processor the WCET of tasks can be analyzed. In former work we explored SCJ level 0 on a multicore version of JOP [27]. The current SCJ specification does not allow a level 0 application on multicores, as synchronization between multiple, truly concurrent, handlers on several cores undermines the simplicity of a cyclic executive. However, in [27] we used static scheduling of the application including the constraints due to resource sharing. Therefore, we are confident that a cyclic executive on a multicore platform is an attractive SCJ execution environment.

JOP is distributed in open source and includes the WCET analysis tool WCA [41] and a method inlining optimizer [11] working at bytecode level. JOP has been ported to many different FPGA platforms and has been used in various industrial applications [35] and research projects.

4.2 The Hardware near Virtual Machine

The Hardware near Virtual Machine (HVM) [14] is a low footprint software VM intended for resource constrained platforms. It is a Java-to-C compiler, but supports interpretation as well, and a mix of the two execution styles. It produces self-contained ANSI-C compatible C code that can be compiled with most embedded compilers and included in existing C based embedded source trees, thus facilitating an incremental addition of Java software components. The HVM interpreter is implemented in a time-predictable manner, thus enabling WCET analysis of Java programs executed on the HVM.

The "write-once, run anywhere" principle of Java can be a significant advantage for embedded developers, as it enables development and debugging of large parts of a projects code base on a PC host platform. The debugging facilities tend to be better and the turn-around times shorter in the context of a desktop operating system. It is clear that real-time issues and hardware interaction can only be executed and debugged on the device itself, but this may be a smaller part of the full code base. To facilitate the principle of doing as much development as possible on a hosted platform a design goal of the HVM is to enable the use of standard Java libraries like OpenJDK and the Java libraries from Oracle. The HVM supports the components of a standard JDK, or indeed any JDK. It is still possible to use special purpose libraries targeted at the specific embedded device. There is a challenge here for the developer to only include minimal parts of standard JDKs that can run on the embedded device. For example, writing to files on a desktop host that contains file system makes perfect sense, but may not be possible on a resource constrained embedded device.

The HVM has been used as execution environment to experiment with various ideas on resource constrained devices. As an example, the concept of hardware objects and 1st level interrupt handlers [40] is supported by the HVM. Research within WCET and schedulability analysis has also used the HVM as execution platform [18, 19]. Finally the development of a Technology Compatibility Kit (TCK) for SCJ Level 0 + 1 runs on top of the HVM (see Section7).

Currently work is ongoing to add support of Java 8 and implementation of SCJ Level 2.

5. SCJ IMPLEMENTATIONS

We have two different JVMs and as well two different implementations of SCJ in Java on top of those JVMs. Some concepts are shared between the two implementations, but they are mostly independent.

Both SCJ implementations use s single, implementation private, class [37] that supports all SCJ based memory areas: immortal memory, mission memory, and private memory. Those classes form a nesting hierarchy and each memory area can be reserved from the parent memory.

Both JVMs support first level interrupt handlers [39]. Such a first level interrupt handler is attached to a programmable timer interrupt, which is then simple the fixed-priority preemptive scheduler for SCJ handlers. A first level interrupt handler can also be used to release an AEH.

5.1 SCJ on JOP

Our SCJ implementation on JOP [42] is coded in Java. There is no underlying operating system and hence functionality typically provided by the operating system (e.g, scheduling, priority inversion control) and access to low-level features (e.g., memory and JVM structures) are provided by JOPs runtime. We have implemented level 0 and level 1 of SCJ.

To implement SCJ's concurrency model, we reuse JOP's real-time threading API [33]. In JOP, real-time activities are supported through two classes, namely `RtThread` and `SwEvent` whose execution is done according to their priority. The `RtThread` class enables the implementation of periodic activities and the `SwEvent` class is used for aperiodic activities that need to be explicitly released by a calls to their `fire()` method. It is therefore natural to implement SCJ's PEHs and AEHs with `RtThread`s and `SwEvent`s respectively.

JOP's real-time threads are executed under the control of a fixed-priority preemptive scheduler. Every thread is assigned a unique priority in order to avoid FIFO queues within priorities. Executing a synchronized method or statement disables all interrupts, including the timer interrupt that triggers the scheduler. In this way, critical sections are executed at the highest possible priority of all threads, thus effectively implementing a priority ceiling protocol where the ceiling is set to the maximum priority that any thread can possibly have.

On the multicore version of JOP each core executes its own scheduler. Handlers are pinned to a core at initialization phase and cannot migrate to a different core. This implements the partitioned scheduler for multicore platforms as defined by SCJ.

SCJ's memory API is implemented using a single system class `Memory`, as described in [37]. SCJ's `ManagedMemory` class delegates functionality to that system class. All the memory-related classes inherited from the RTSJ are empty classes with the exception of the `ImmortalMemory` class, which

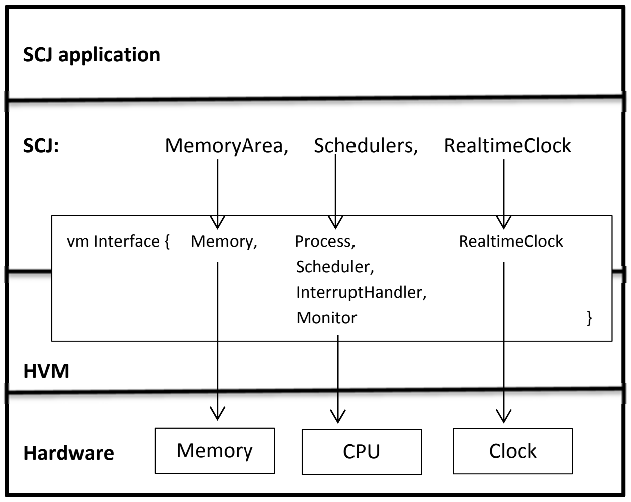

Figure 1: SCJ architecture with the vm Interface to HVM.

however also delegates to the system's `Memory` class.

Our current implementation still has following restrictions: (1) we do not yet implement the `OneShotEventHandler` class, (2) only a variation of priority ceiling emulation, where all lock objects have the maximum priority, is implemented, (3) happenigs and POSIX-related classes are not needed in our embedded platform, and (4) we do not yet fully support user-defined clocks [51]. An evaluation of our implementation can be found in [32].

5.2 SCJ on HVM

The SCJ implementation on top of HVM [14, 45] has the architecture, shown in Figure 1, with a minimal hardware interface layer specified in the `vm` Interface.

This interface is divided into three parts:

- `Memory` that controls the memory allocation;
- `Process`, `Scheduler`, `InterruptHandler`, and `Monitor` that define the interface to process, process scheduling, context switch, and synchronization; and
- `RealtimeClock` that defines clock specific methods.

The SCJ implementation on HVM is a bare metal implementation, which takes advantages of hardware objects and other hardware near features. Thus, it has no native function layer. Currently, level 0 and level 1 have been implemented, because they target applications running on resource constrained embedded platforms.

The implementation strategy for the SCJ classes that use classes from the `vm` Interface is the same. Here delegation is used, e.g.,

- class `MemoryArea` has a field `delegate` of type `vm.Memory`,
- class `ManagedEventHandler` uses an infrastructure class `ScjProcess` that has a field `delegate` of type `vm.Process`, and
- class `PriorityScheduler` delegates to an infrastructure class `PrioritySchedulerImpl` that implements `vm.Scheduler`.

SCJ supports method synchronization for shared resources. This is implemented in an SCJ infrastructure class `Monitor` which extends the abstract `vm.Monitor` class. This infrastructure class is also used for implementing the priority ceiling emulation that SCJ requires to avoid priority inversion.

6. ANALYSIS TOOLS

During the CJ4ES project, two tools have been developed to support programmers developing for the SCJ specification. In particular, both tools aim at helping programmers with some of the issues arising from the scoped memory model of SCJ: avoiding runtime exceptions due to memory safety violations and providing a sound estimate of the maximum amount of memory needed for a given SCJ application. The former is essential for certification of safety-critical applications and the latter is needed for proper sizing of memory scopes.

Both tools are based on the T.J. Watson Libraries for Analysis (WALA) [50]. WALA provides highly scalable and extensible libraries for several state-of-the-art pointer analyses for Java bytecode, including user definable context sensitive analyses, as well as good integration with the Eclipse development environment.

6.1 Memory Safety

The scoped memory model, as described in Section 3, allows programmers to store references to an object in another scope. This potentially introduces dangling pointers to objects that no longer exist. There are several approaches for dealing with this problem, the two main approaches being annotations and static analysis. In the former approach, annotations are added to key elements of the program, indicating the intended scope for those elements. Based on the annotations, an automated tool can then verify that the indicated scope use will not lead to a memory safety violation [20, 49]. The latter approach does not require, but may make use of, programmer defined annotations in the application. Instead a *memory safety analysis* is performed over the entire program, in order to verify that no reference can ever point to an object in a shorter lived memory area and thus, that no memory safety violations can occur in the program.

In spirit with the compiler mantra to provide feedback to programmers as soon as possible we have followed the latter approach described above, and implemented a memory safety analysis that is able to identify illegal memory assignments (assignments that may lead to a dangling reference) [9]. The analysis was designed as a special context-sensitive points-to analysis, using memory scopes as contexts. Using the built-in points-to analysis in the WALA framework [50], which can be parameterised over user-defined contexts, it was relatively straightforward to implement our analysis. The analysis is sound but may report false positives, i.e., false warnings of potential memory safety violations. However, on the suite of applications the analysis was tested on, only few false positives were reported.

In addition to statically verifying that a program cannot violate memory safety, the results of our memory safety analysis can also be used for generating scope annotations that can be added to the program. This is useful, e.g., for documenting the results of the analysis in a fashion that can be automatically checked by other tools [20,49], or even be used for understanding a program developed by a third party.

Our analysis is closely related to the data-flow analysis developed by Siebert [44] for the RTSJ, but exploits the simpler structure of the SCJ memory model (see Section 3.2) to achieve both high precision and good performance. A hardware implementation of the scope checking was presented by Rios et al. [31], showing that such checking can be done very efficiently for SCJ in hardware due to the simple stack structure of the memory scopes in use.

6.2 Worst-Case Memory Consumption

As described in Section 3.2, the programmer has to explicitly provide memory bounds for the backing store needed, e.g., for event handlers and memory areas in general. Manually calculating these bounds is, in general, difficult and error prone, since the programmer has to take all possible program paths into account while also keeping track of the current scope stack and map all of this onto a platform with particular memory usage and requirements for alignment, etc.

To help the programmer with the task of calculating memory bounds, we have developed a fully automated *worst-case memory consumption analysis* [2]. Our analysis is similar in spirit to the analysis described in [25], but has been adapted to the simpler memory model of SCJ. Similar to [25] the tool is also based on the well-known IPET method used for transforming WCET analysis into an integer linear programming problem. However, the tool is based on the SCJ memory model described in [37] and also shares some commonality with the tool for memory safety analysis described in the previous subsection; in particular the memory consumption analysis was also implemented in the WALA framework [50] combined with an ILP solver.

7. TESTING

A Java specification request must include a Technology Compatibility Kit (TCK) that contains a suite of tests checking whether an implementation conforms to the specification. Also, a TCK is supposed to have tools to run the tests and report the results. Since the project aims at certifiable applications, tests are crucial for achieving the results. Without thorough testing, there is little chance of an application being certified. Of course this testing is for the application as a whole, but the software platform on which it relies should be tested as well.

For SCJ, we have seen one previous proposal for a TCK [53]; it is from Purdue University and treats a somewhat dated version of SCJ. It uses a top down approach and tests central concepts by executing a number of missions initiated from a `Safelet` interface. We aim at a bottom up approach, where individual classes are subjected to unit tests. Thus the two may be aligned to complement each other.

Essentially, it is not complicated to write tests. Consider for example a test of `RelativeTime add (long millis, int nanos)`, it could be the simple program:

```
rel = new RelativeTime();
res =  rel.add(0,1000001);
```

It may be tedious to write tests for the different interesting argument values, but it is even more tedious to compute the result and check it with an assertion. In this case:

```
assert res.getMilliseconds()==1
    && res.getNanoseconds()==0;
```

An alternative is to give general specification of the result as a post-condition in the Java Modeling Language (JML) [16] which annotates the Java source code for the method:

```
ensures \result != null
ensures \result.getMilliseconds() - getMilliseconds()
           - millis
          + (\result.getNanoseconds()
               - getNanoseconds() - nanos)/1000000
          == 0;
ensures (\result.getNanoseconds()-getNanoseconds()
          - nanos ) % 1000000 == 0;
```

With suitable test execution tools, this will do all the checking needed for the result. This approach with testing supported by formal specifications is further elaborated in [28].

Besides the effort saved in checking concrete results, the JML specifications could form the basis for actually formally verifying an implementation of the SCJ profile. This would add much assurance to the trustworthiness of the implementation.

8. HARDWARE SUPPORT

As we use one JVM as a Java processor in an FPGA we also explored possibilities to enhance that processor with hardware units to support SCJ specific functions. The intention is to perform operations faster and especially more time-predictable in hardware. We explored pointer assignment checks for scopes and locking support for a multicore version of JOP.

8.1 Scope Checks

As mentioned in Section 6.1, the use of the scoped memory model can lead to dangling pointers. As a consequence, the JVM must check the referential integrity by ensuring that objects allocated in a memory area only store references to objects allocated either in the same or in a longer-lived memory area. Enforcing this referential integrity on every reference assignment becomes a source of execution time overhead for an application.

Given the simplified memory model of SCJ, the scope nesting level (see Section 3.2) can be used to check the legality of every reference assignment. In JOP's SCJ implementation, the scope level of where an object is allocated is associated with the object itself when the object is created. This information is stored in the object's reference and recovered during the execution of the `putfield`, `putstatic`, and `aastore` bytecodes, i.e., the reference assignment bytecodes.

Reference assignment checks require only a simple comparison between the scope levels of the source and destination objects and this comparison can be efficiently performed in hardware. The scope level information is available during the execution of the mentioned bytecodes and therefore the check itself is included in JOP's memory management unit (MMU) as part of the execution of those bytecodes. The check shall only be done when it is an assignment of a reference, so the hardware needs to know if the operation involves a reference or a primitive data type. Such information is available on special versions of the `putfield/putstatic` bytecodes and in the `aastore` bytecode. A new microcode instruction is used to signal to the MMU that a reference assignment will take place and the scope nesting levels need to be checked. An interrupt is generated inside the MMU when an illegal assignment occurs. This flag is used to throw an `IllegalAssignmentError`.

The overhead of this operation is of one clock cycle compared to the execution of the mentioned bytecodes without the reference assignment checks. This hardware implemented scope check is around 14 times faster than a software solution. There is an extra timing overhead that comes from adding the scope level information at object creation time. However, object creation is an operation with low frequency of execution [34]. The hardware cost is of approximately a 4% increase in the MMU size.

8.2 Multi-Core Locking Unit

In Java, every object can serve as a lock, either explicitly through a synchronized block or implicitly using an object's synchronized methods. This is also the case in SCJ, although synchronized blocks are forbidden. Java locks are usually implemented in the JVM using an underlying synchronization mechanism, such as compare-and-swap (CAS). In JOP a global lock is used, which in effect reduces all individual locks to a single global lock. If locks are implemented using CAS they are not reduced to a single lock, however CAS uses shared memory, which leads to memory arbitration. Furthermore to track locks either a software hash-map is used, which is inappropriate for real-time systems, or potential lock objects need an extra header field, increasing objects size. This inspired our work on a multi-core hardware locking-unit.

Our first attempt [47] uses a content-addressable memory (CAM) to store the address of an object used as a lock. Whenever a core supplies an object's address it is checked against all existing entries. In the case where the entry already exists, the core is enqueued and blocked until the lock becomes available. Otherwise a new entry is created for the supplied address and the core is allowed to continue execution.

Using the CAM enables the address-to-queue mapping to be done without the need for a lock entry in objects or the need for a software hash-map. However, the queuing is still done in software, which means that this solution still suffers from memory arbitration.

Our second attempt merges the CAM, the queues, and the global lock into a single unit. Moving the queues to hardware enables the unit to be accessed through core local microcode, instead of Java code in shared memory. This removes any memory arbitration in the locking procedure. Serializing lock requests is still necessary, however merging the global lock with the unit means that only a single unit has to be accessed, reducing the number of software steps.

9. SCOPE USAGE AND LIBRARIES

Correct use of the scoped memory model is perhaps SCJ's most difficult feature to use. Passing arguments and returning results without the use of static fields is not obvious. In [30] we analyzed the expressive power of SCJ's memory model and proposed patterns for its safe use. We provided a collection of seven scoped-memory use patterns specific to SCJ that vary in complexity and that can be used for simple subroutines, sequences of subroutine calls, and nested calls. The patterns avoid memory leaks, unnecessary copying of values, and are illustrated with an implementation in the SCJ profile.

In addition to scope-aware patterns, we explored the topic of scope-aware Java libraries. In [29] we have: (1) identified programming patterns and idioms in the standard Java class libraries that make them not suitable for SCJ, (2) provided different ways to mitigate the impact of the identified problematic patterns, and (3) implemented representative scope-safe classes with minimal changes in the standard Java libraries. We implemented a total of five scope-safe classes from commonly used libraries having as a starting point the reference implementation of OpenJDK 6 and the restrictions imposed by the already defined class libraries in SCJ [17]. Our developed classes maintain referential integrity (between objects created within the library class), have predictable memory consumption, and predictable worst-case execution time.

Those characteristics were achieved by combining different techniques such as an explicit change of allocation context, restricting the maximum number of elements that internal arrays can have, changing the exit condition in loops (to automate the loop bound detection), and removing constructors where we cannot guarantee the execution time or memory consumption.

We have tested our libraries using a combination of tools. Referential integrity was tested using the private memory analysis tool described in Section 6.1, memory consumption was both measured and analyzed with the tool described in Section 6.2, and the WCET was tested using JOP's WCET analysis tool.

10. APPLICATIONS

To explore the expressiveness of SCJ and also to have test cases, we have implemented several applications and small test cases on top of SCJ.

10.1 RepRap

A lack of SCJ use-cases motivated the development of a RepRap 3D printer in SCJ [48]. A RepRap printer melts plastic and extrudes it in 3 dimensional space according to printing instructions (G-codes). This enables it to construct or "print" 3D objects in plastic. Figure 2 shows the RepRap hardware, an FPGA board containing JOP, and the costume made interface board between the FPGA board and the electronics of the RepRap printer.

The printer uses stepper motors for movement and a resistor as a heating element. We consider the printer a safety-critical application as it employs physical movement and heating above 200°C.

The system consists of the physical printer, an electronic interface board, a SCJ RepRap controller running on JOP on an FPGA, and a PC. The PC generates printing instructions from a 3D image that are sent over a serial line to the controller. The controller enqueues and executes the instructions and controls the motors and the heating element in the process through the interface board.

The controller consists of the periodic event handlers HostController, CommandParser, CommandController, and RepRapController. The HostController handles all communication with the host (PC). When the HostController sees an instruction delimiter, the previously received characters are sent to the CommandParser, which parses the instruction and enqueues the corresponding Command object in the CommandController. The CommandController executes the commands in FIFO order, with some commands modify-

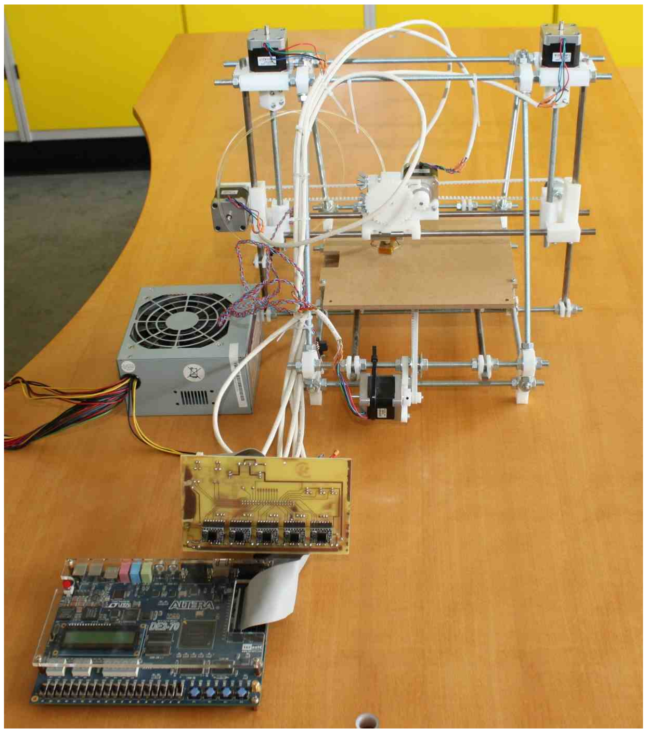

Figure 2: RepRap setup without the host

ing control parameters in the RepRapController and others writing back to the host through the HostController. The RepRapController controls the motors and temperature according to control parameters. The motors are stepped towards a position in a timed manner, ensuring that all motors reach their goal at the same time regardless of the distance.

10.2 CSP Watchdog

Another application developed to test our implementation is a CSP-based watchdog [2]. CSP stands for Cubesat space protocol, which is a network-layer protocol developed at Aalborg University and used in small space-research satellites called Cubesats. This watchdog application uses two PEHs and one managed interrupt service routine (MISR). One of the PEHs is in charge of sending ping packets to a set of CSP nodes while the other functions as a router, distributing CSP packets from its packet queue to an appropriate destination. The MISR is triggered upon reception of a complete CSP packet, processes the incoming packet (e.g., reads relevant fields of the CSP header), and puts the packet into the router queue for later distribution. Connecting our JOP-based SCJ implementation to an on-board Cubesat computer board, provided by GomSpace ApS, tested the application.

10.3 External Applications

Within the project we also searched for and explored SCJ example applications from other research groups. When needed, we adapted them to the current version of the specification. The updated code is available at the project's repository.

10.3.1 miniCDj

The miniCDj benchmark is a reduced version of the CDx benchmark that is described in [13]. This benchmark generates simulated radar frames containing airplane positions and calculates possible collisions between those simulated radar frames. We have used two versions of it: the original implementation which uses a cyclic-executive (level 0), and a parallel version (level 1) [52] adapted at the University of York, which divides the task of looking for collisions into a fixed number of AEHs. In both cases we have updated the benchmarks to the current version of the SCJ specification as both of them were based on version 0.76.

10.3.2 Purdue's SCJ TCK

The technology compatibility kit described in [53] is an early work done to develop a TCK for SCJ. It is however based on version 0.76 of the specification. We have updated that TCK to the current version and used it to test the level 0 and level 1 features of the SCJ implementation of JOP.

11. CONCLUSIONS

The paper summarizes the research and development work of the three-year project Certifiable Java for Embedded Systems (CJ4ES). With the project we developed two independent versions of safety-critical Java (SCJ) on two different Java virtual machines. We explored and assessed SCJ by investigating design patterns for libraries under the scoped memory model of SCJ, implementing whole applications on top of SCJ, and developing tools for scope usage and a test framework for SCJ implementations. All results are provided in open-source to enable exploration of our results and simplify future cooperation. One of the questions of CJ4ES was, if SCJ is suitable for future safety-critical applications. Our conclusion is that SCJ is not easy to use for programmer trained in general purpose Java programming. However, in our opinion exactly those restrictions within SCJ, which make it hard to use in the first place, make SCJ a good choice for building future safety-critical applications with a safe programming language.

Acknowledgment

This work is part of the project "Certifiable Java for Embedded Systems" (CJ4ES) and has received partial funding from the Danish Research Council for Technology and Production Sciences under contract 10-083159.

Source Access

The two presented JVMs and the SCJ implementations on top of them are available in open source. The Java processor JOP and the SCJ implementation are available from GitHub at `https://github.com/jop-devel/jop`. The hardware near virtual machine is available from `http://www.icelab.dk/download.html`, which includes a version of SCJ as well. The source of SCJ for HVM is also available from GitHub at `https://github.com/scj-devel/hvm-scj`.

12. REFERENCES

[1] aJile. aj-100 real-time low power Java processor. preliminary data sheet, 2000.

[2] J. L. Andersen, M. Todberg, A. E. Dalsgaard, and R. R. Hansen. Worst-case memory consumption analysis for SCJ. In *Proceedings of the 11th*

International Workshop on Java Technologies for Real-time and Embedded Systems, pages 2–10. ACM Press, 2013.

[3] Aonix. Perc pico 1.1 user manual. http://research.aonix.com/jsc/pico-manual.4-19-08.pdf, April 2008.

[4] A. Armbruster, 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. *Trans. on Embedded Computing Sys.*, 7(1):1–49, 2007.

[5] T. Bøgholm, R. R. Hansen, A. P. Ravn, B. Thomsen, and H. Søndergaard. A predictable java profile: rationale and implementations. In *JTRES '09: Proceedings of the 7th International Workshop on Java Technologies for Real-Time and Embedded Systems*, pages 150–159, New York, NY, USA, 2009. ACM.

[6] T. Bogholm, H. Kragh-Hansen, P. Olsen, B. Thomsen, and K. G. Larsen. Model-based schedulability analysis of safety critical hard real-time Java programs. In *Proceedings of the 6th international workshop on Java technologies for real-time and embedded systems (JTRES 2008)*, pages 106–114, New York, NY, USA, 2008. ACM.

[7] 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.

[8] 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.

[9] A. E. Dalsgaard, R. R. Hansen, and M. Schoeberl. Private memory allocation analysis for safety-critical Java. In *Proceedings of the 10th International Workshop on Java Technologies for Real-Time and Embedded Systems (JTRES 2012)*, pages 9–17, Copenhagen, DK, October 2012. ACM.

[10] T. Henties, J. J. Hunt, D. Locke, K. Nilsen, M. Schoeberl, and J. Vitek. Java for safety-critical applications. In *2nd International Workshop on the Certification of Safety-Critical Software Controlled Systems (SafeCert 2009)*, York, United Kingdom, Mar. 2009.

[11] S. Hepp and M. Schoeberl. Worst-case execution time based optimization of real-time Java programs. In *Proceedings of the 15th IEEE International Symposium on Object/component/service-oriented Real-time distributed Computing (ISORC 2012)*, pages 64–70, Shenzhen, China, April 2012. IEEE.

[12] B. Huber, W. Puffitsch, and M. Schoeberl. Worst-case execution time analysis driven object cache design. *Concurrency and Computation: Practice and Experience*, 24(8):753–771, 2012.

[13] T. Kalibera, J. Hagelberg, F. Pizlo, A. Plsek, B. Titzer, and J. Vitek. Cdx: a family of real-time java benchmarks. In *JTRES '09: Proceedings of the 7th International Workshop on Java Technologies for Real-Time and Embedded Systems*, pages 41–50, New York, NY, USA, 2009. ACM.

[14] S. Korsholm, H. Søndergaard, and A. Ravn. A real-time java tool chain for resource constrained platforms. *Concurrency and Computation: Practice & Experience*, 2013:1–25, September 2013.

[15] 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.

[16] Leavens. The Java Modeling Language (JML). www.eecs.ucf.edu/l̃eavens/JML/index.shtml, Visited June 2014.

[17] D. Locke, B. S. Andersen, B. Brosgol, M. Fulton, T. Henties, J. J. Hunt, J. O. Nielsen, K. Nilsen, M. Schoeberl, J. Tokar, J. Vitek, and A. Wellings. Safety-critical Java technology specification, public draft, 2011.

[18] K. S. Luckow, T. Bøgholm, and B. Thomsen. Supporting development of energy-optimised java real-time systems using tetasarts. In *Work-in-Progress Proceedings of the 19th Real-Time and Embedded Technology and Application Symposium*, 2013.

[19] K. S. Luckow, T. Bøgholm, B. Thomsen, and K. G. Larsen. Tetasarts: A tool for modular timing analysis of safety critical java systems. In *Proceedings of the 11th International Workshop on Java Technologies for Real-time and Embedded Systems*, JTRES '13, 2013.

[20] K. Nilsen. A type system to assure scope safety within safety-critical java modules. In *Proceedings of the 4th international workshop on Java technologies for real-time and embedded systems*, JTRES '06, pages 97–106, New York, NY, USA, 2006. ACM.

[21] K. Nilsen. Harmonizing alternative approaches to safety-critical development with Java. In *Proceedings of the 9th International Workshop on Java Technologies for Real-Time and Embedded Systems (JTRES 2011)*, pages 54–63, 2011.

[22] K. Nilsen and S. Lee. Perc real-time api (draft 1.3). newmonics, July 1998.

[23] C. Pitter and M. Schoeberl. A real-time Java chip-multiprocessor. *ACM Trans. Embed. Comput. Syst.*, 10(1):9:1–34, 2010.

[24] A. Plsek, L. Zhao, V. H. Sahin, D. Tang, T. Kalibera, and J. Vitek. Developing safety critical Java applications with oSCJ/L0. In *Proceedings of the 8th International Workshop on Java Technologies for Real-Time and Embedded Systems (JTRES 2010)*, pages 95–101, New York, NY, USA, 2010. ACM.

[25] W. Puffitsch, B. Huber, and M. Schoeberl. Worst-case analysis of heap allocations. In *Proceedings of the 4th International Symposium On Leveraging Applications of Formal Methods, Verification and Validation (ISoLA 2010)*, 2010.

[26] P. Puschner 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.

[27] A. P. Ravn and M. Schoeberl. Safety-critical Java with cyclic executives on chip-multiprocessors. *Concurrency and Computation: Practice and Experience*, 24:772–788, 2012.

[28] A. P. Ravn and H. Søndergaard. A test suite for Safety-Critical Java using JML. In *Proceedings of the 11th International Workshop on Java Technologies for*

Real-time and Embedded Systems, JTRES '13, pages 80–88, New York, NY, USA, 2013. ACM.

[29] J. Rios and M. Schoeberl. Reusable Libraries for Safety-Critical Java. In *Object/Component/Service-Oriented Real-Time Distributed Computing (ISORC), 2014 IEEE 17th International Symposium on*, pages 188–197, June 2014.

[30] J. R. Rios, K. Nilsen, and M. Schoeberl. Patterns for safety-critical Java memory usage. In *Proceedings of the 10th International Workshop on Java Technologies for Real-time and Embedded Systems*, page 1–8, 2012.

[31] J. R. Rios and M. Schoeberl. Hardware support for safety-critical Java scope checks. In *Proceedings of the 15th IEEE International Symposium on Object/component/service-oriented Real-time distributed Computing (ISORC 2012)*, pages 31–38, Shenzhen, China, April 2012. IEEE.

[32] J. R. Rios and M. Schoeberl. An evaluation of safety-critical Java on a Java processor. In *Proceedings of the 10th Workshop on Software Technologies for Embedded and Ubiquitous Systems (SEUS 2014)*, Reno, Nevada, USA, June 2014.

[33] 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. IEEE.

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

[35] M. Schoeberl. Application experiences with a real-time Java processor. In *Proceedings of the 17th IFAC World Congress*, pages 9320–9325, Seoul, Korea, July 2008.

[36] M. Schoeberl. A Java processor architecture for embedded real-time systems. *Journal of Systems Architecture*, 54/1–2:265–286, 2008.

[37] M. Schoeberl. Memory management for safety-critical Java. In *Proceedings of the 9th International Workshop on Java Technologies for Real-Time and Embedded Systems (JTRES 2011)*, pages 47–53, York, UK, September 2011. ACM.

[38] M. Schoeberl. A time-predictable object cache. In *Proceedings of the 14th IEEE International Symposium on Object/component/service-oriented Real-time distributed Computing (ISORC 2011)*, pages 99–105, Newport Beach, CA, USA, March 2011. IEEE Computer Society.

[39] M. Schoeberl, S. Korsholm, T. Kalibera, and A. P. Ravn. A hardware abstraction layer in Java. *ACM Trans. Embed. Comput. Syst.*, 10(4):42:1–42:40, November 2011.

[40] M. Schoeberl, S. Korsholm, C. Thalinger, and A. P. Ravn. Hardware objects for Java. In *Proceedings of the 11th IEEE International Symposium on Object/component/service-oriented Real-time distributed Computing (ISORC 2008)*, pages 445–452, Orlando, Florida, USA, May 2008. IEEE Computer Society.

[41] M. Schoeberl, W. Puffitsch, R. U. Pedersen, and B. Huber. Worst-case execution time analysis for a Java processor. *Software: Practice and Experience*, 40/6:507–542, 2010.

[42] M. Schoeberl and J. R. Rios. Safety-critical Java on a Java processor. In *Proceedings of the 10th International Workshop on Java Technologies for Real-Time and Embedded Systems (JTRES 2012)*, pages 54–61, Copenhagen, DK, October 2012. ACM.

[43] M. Schoeberl, 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)*, pages 94–101, Santorini Island, Greece, May 2007. IEEE Computer Society.

[44] 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.

[45] H. Søndergaard, S. E. Korsholm, and A. P. Ravn. Safety-Critical Java for low-end embedded platforms. In *Proceedings of the 10th International Workshop on Java Technologies for Real-time and Embedded Systems*, JTRES '12, pages 44–53, New York, NY, USA, 2012. ACM.

[46] H. Søndergaard, 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*, page 38–47. ACM, 2006.

[47] T. B. Strøm, W. Puffitsch, and M. Schoeberl. Chip-multiprocessor hardware locks for safety-critical Java. In *Proceedings of the 11th International Workshop on Java Technologies for Real-Time and Embedded Systems (JTRES 2013)*, pages 38–46, Karlsruhe, DE, October 2013. ACM.

[48] T. B. Strøm and M. Schoeberl. A desktop 3d printer in safety-critical Java. In *Proceedings of the 10th International Workshop on Java Technologies for Real-Time and Embedded Systems (JTRES 2012)*, pages 72–79, Copenhagen, DK, October 2012. ACM.

[49] D. Tang, A. Plsek, and J. Vitek. Static checking of safety critical java annotations. In *Proceedings of the 8th International Workshop on Java Technologies for Real-Time and Embedded Systems*, JTRES '10, pages 148–154, New York, NY, USA, 2010. ACM.

[50] T.J. Watson libraries for analysis (WALA). `http://wala.sf.net/`.

[51] A. Wellings and M. Schoeberl. User-defined clocks in the real-time specification for Java. In *Proceedings of the 9th International Workshop on Java Technologies for Real-Time and Embedded Systems (JTRES 2011)*, pages 74–81, York, UK, September 2011. ACM.

[52] F. Zeyda, A. Cavalcanti, A. Wellings, J. Woodcock, and K. Wei. Refinement of the Parallel CDx. Technical report, University of York, 2012.

[53] L. Zhao, D. Tang, and J. Vitek. A technology compatibility kit for safety critical java. In *Proceedings of the 7th International Workshop on Java Technologies for Real-Time and Embedded Systems*, JTRES '09, pages 160–168, New York, NY, USA, 2009. ACM.