Safety-Critical Java on a Java Processor

Martin Schoeberl
Department of Informatics and Mathematical Modeling
Technical University of Denmark
masca@imm.dtu.dk

Juan Ricardo Rios
Department of Informatics and Mathematical Modeling
Technical University of Denmark
jrri@imm.dtu.dk

ABSTRACT

The safety-critical Java (SCJ) specification is developed within the Java Community Process under specification request number JSR 302. The specification is available as public draft, but details are still discussed by the expert group. In this stage of the specification we need prototype implementations of SCJ and first test applications that are written with SCJ, even when the specification is not finalized. The feedback from those prototype implementations is needed for final decisions. To help the SCJ expert group, a prototype implementation of SCJ on top of the Java optimized processor is developed and presented in this paper. This implementation raises issues in the SCJ specification and provides feedback to the expert group.

Categories and Subject Descriptors

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

Keywords

real-time systems; Java; safety-critical systems

1. INTRODUCTION

Safety-critical systems are currently developed in Ada and in C. However, Ada has become a niche language and C can be considered a very low-level language. Currently there is effort to establish Java as a language to build safety-critical systems. The arguments are that Java is safer than C, the wide usage of Java led to mature tools (e.g., compiler), and many programmers know Java very well.

However, standard Java is probably too complex to build certifiable safety-critical systems. Therefore, a subset of Java, called safety-critical Java (SCJ), is specified under the Java Community Process [8, 6]. The specification is currently in public draft. To evaluate the specification we need independent implementations and application use cases. This paper describes a prototype implementation of SCJ on top of the Java processor JOP [17]. As the specification is a moving target the implementation is under continuous development.

The implementation is open-source and can, although targeted for JOP, also be executed on a standard PC with a software simulation of JOP. Therefore, it can be used to evaluate the SCJ and also to experiment with additional or different features that are not (yet) officially part of the specification. We used an early version of this implementation for example to experiment with active clocks that can drive scheduling events [26]. SCJ applications and examples are already emerging [23]. One example, a 3D printer controller [25], already uses the SCJ implementation that is described in this paper.

The main contribution of this work is an open-source implementation of SCJ. Furthermore, we present some implementation issues (and workarounds) that arise from the Java package crossing of RTSJ and SCJ classes and propose a simplification of aperiodic event handlers in SCJ.

The paper is organized as follows: the following section presents related work on real-time Java subsets and first implementations of SCJ. In Section 3 we provide a brief overview of the SCJ specification. This section contains also a complete, minimal SCJ *Hello World* example. Details of the implementation of SCJ on top of JOP are given in Section 4. Suggestions for changes in the SCJ specification are discussed in Section 5 and the paper is concluded with Section 6.

2. RELATED WORK

Java for real-time systems started with work on PERC Pico [9]. Later on, the Real-Time Specification for Java (RTSJ) [4] 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 [13]. 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`, 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 [7] to emphasize the heritage of the concepts from the Ada

Ravenscar tasking profile [5]. Later proposals for a safety-critical Java profile argue for an API that is independent of the RTSJ [16, 22]. 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 [3].

As the SCJ specification is still in draft, implementations of it are rare. A first prototype of a level 0 implementation on top of OVM [2] and Fiji [11] is presented in [12]. A SCJ like implementation, called predictable Java profile [3], implements a different class hierarchy where RTSJ classes extend SCJ classes. A SCJ implementation on top of a JVM for very memory constrained devices is presented in [24]. It is reported that a simple SCJ application can execute in 35 KB ROM and just 10 KB RAM.

The upcoming version of PERC Pico [1] 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 [10]. 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) [8] is intended for future safety-critical systems that need certification. The SCJ specification is developed within the Java community process under specification request number JSR 302. To allow certification of Java programs only a very restricted subset of Java is defined. SCJ itself is based on the RTSJ [4]. It is a subset of RTSJ with some additional class files. It shall be possible to provide the reference implementation of SCJ on top of a standard RTSJ.

To cover different criticality levels, SCJ defines three different levels with increasing expressive power for the application programmer, but also increasing complexity of implementation and certification.

Level 0 admits a sequence of missions, where each mission consists of periodic handlers under the control of a single-threaded cyclic executive. The level is intended as a stepping-stone for developers that are using cyclic executives, programmed in C or Ada. The concurrency model stays the same, only the language changes. Level 1 introduces a preemptive scheduler, very similar to the Ada Ravenscar tasking profile [5]. Level 1 also supports a sequence of missions. Level 2 introduces nested missions and supports an adapted version of RTSJ's `NoHeapRealtimeThread`. The nested missions of level 2 allow more dynamic systems, where threads can be started and stopped, while an outer mission continues to execute.

With respect to memory areas, all three levels support the memory model with three layers: immortal memory, mission memory, and handler or thread private scopes. The only difference between the levels is that all handlers in level 0 can use the same backing store for their private memories.

Concurrency is represented as *handlers* in SCJ, similar to RTSJ style event handlers. In fact the SCJ handlers are a subclass of RTSJ's `BoundAsyncEventHandler`. These handlers are either periodic or event triggered.

3.1 Missions and Schedulable Objects

An application may have several missions. A mission consists of a scoped memory, the mission memory, and a set of managed handlers. A level 2 mission may also contain managed threads.[1] The notion of *managed* handlers and threads means that the start and termination of those entities is under the control of the SCJ implementation. The handlers within a mission are created during mission initialization and the number of handlers is fixed for a mission. Handlers come in two flavors: a periodic event handler to be released time-triggered and an aperiodic handler released by an event. The event to release an aperiodic handler can be a software event or an interrupt. The handlers and threads are also called *schedulable objects*.

A mission consists of three phases: initialization, execution, and cleanup. At the initialization phase the mission memory is created by the SCJ implementation. Mission memory is entered and the handlers and threads are created within the mission memory. Furthermore, all data structures that are needed for the handlers to communicate need to be allocated in mission memory (or in immortal memory). Only immortal and mission memory is shared between threads.

On the transition to the execution phase all handlers are *started*. During the execution phase no new handlers can be registered or started. In the execution phase temporary objects are allocated in handler private memory. Allocation in mission memory or immortal memory is not prohibited. However, allocating data in those memories can result in a memory leak. After the cleanup phase, the mission memory is cleared and a new mission can be started in a new, possible differently sized, mission memory.

A SCJ application is represented by a class that implements `Safelet` and at least one class that extends `Mission`. Simple programs, consisting of a single mission, can use one class that extends `Mission` and implements `Safelet`. How this *main* class is specified as the SCJ application is vendor specific.

3.2 The Memory Model

SCJ simplifies the RTSJ memory classes and introduces managed memories. As there is no garbage collector in SCJ, heap memory is nonexistent. SCJ supports RTSJ style immortal memory. RTSJ scoped memories are available in two special versions: mission memory and private memory, which extend the common superclass `ManagedMemory`. Mission memory is a memory area that is alive as long as a mission is executing. Therefore, it is used for data that is shared among handlers.

For temporary objects, each handler has a private memory. This private memory is entered on the handler release and exited after finishing the release. A handler can also enter a nested private memory with `enterPrivateMemory()`. Conceptionally, these private memories are anonymous as `enterPrivateMemory()` is a static method on the SCJ class

[1]A managed thread, as it is a RTSJ real-time thread, needs to invoke `waitForNextPeriod()` to finish the current release.

```
public class HelloSafelet implements Safelet {

   @Override
   public MissionSequencer getSequencer() {
      return new HelloSequencer(new HelloMission());
   }

   @Override
   public long immortalMemorySize() {
      return 1000;
   }
}
```

Figure 1: A `Safelet` defines the application.

```
public class HelloSequencer extends MissionSequencer {

   Mission m;

   public HelloSequencer(Mission mission) {

      super(new PriorityParameters(13),
           new StorageParameters(1000000, null));
      m = mission;
   }

   @Override
   protected Mission getNextMission() {
      return m;
   }
}
```

Figure 2: A simple, application specific mission sequencer.

`ManagedMemory`. However, with `getCurrentManagedMemory()` a SCJ application can get a reference to it.

The memory areas of SCJ form a clear hierarchy and an index can be assigned to each area: starting with index 0 for immortal memory, index 1 for the initial mission memory, index 2 and upper for private memories and nested mission memories. These levels simplify the assignment checks, which are needed to avoid dangling references to objects with a shorter lifetime [14].

3.3 A Level 1 Example

In the following we show a SCJ *Hello World* example, where each component is defined by an application specific class.

Figure 1 shows the application defining class, which implements `Safelet`. An instance of this class is allocated in immortal memory. How this instance is defined and how the class is instantiated is *implementation defined*. When this class is instantiated by the runtime system, it needs to have a parameter-less constructor. When no constructor is defined, Java defines a parameter-less per default. Furthermore, as reflection is not available within SCJ, some JVM magic is needed to convert a class name into a class type. With this class type the application class can be created with `newInstance(Class type)`.

Our example contains an application specific sequencer (see Figure 2), which returns always the same mission on `getNextMission()`. The sequencer is created after the mission

```
public class HelloMission extends Mission {

   @Override
   protected void initialize() {

      OutputStream os = null;
      try {
         os = Connector.openOutputStream("console:");
      } catch (IOException e) {
         throw new Error("No␣console␣available");
      }

      HelloHandler hh = new HelloHandler(
                     new SimplePrintStream(os));
      hh.register();
   }

   @Override
   public long missionMemorySize() {
      return 100000;
   }
}
```

Figure 3: A mission with a single handler.

object is created and both objects are allocated in immortal memory.

Figure 3 shows the example mission. During initialization a reference to the console object is requested and a `SimplePrintStream` is created. Furthermore, one handler is created and registered. These objects are now allocated in mission memory. Therefore, we cannot store the reference to the output stream in a static variable for easy access. All references to shared objects need to be passed via the constructor of the handlers. In our example it is the reference to the console via `SimplePrintStream`.

SCJ does not support standard input and output streams (`System.in` and `System.out`). Instead SCJ defines a console and uses the Java micro edition (J2ME) connector framework.

The active component, a periodic event handler, is shown in Figure 4. The period is set to half a second. Within `handleAsyncEvent()` a string is constructed to print out the iteration number. This string construction generates several objects, which are allocated in private memory.

We can see that an application needs to define the maximum memory consumption of mission memory, immortal memory, and the maximum memory for all private memories a handler may use.

The example SCJ application provides an application-defined class for each needed component. A minimalistic version of the *Hello World* example could be implemented with one named class that implements `Safelet` and extends `Mission`. A utility class from SCJ, `LinearMissionSequencer` can serve as sequencer, and the periodic event handler can be declared as anonymous class within `initialize()` from `Mission`.

4. IMPLEMENTATION

We base our implementation on the Java processor JOP [17] and the available infrastructure for thread scheduling. The original version of JOP supports garbage collection (GC) and no scoped memories. Without a GC, the whole heap implicitly becomes immortal memory. We

```
public class HelloHandler extends PeriodicEventHandler {

   SimplePrintStream out;
   int cnt;

   public HelloHandler(SimplePrintStream sps) {
      super(new PriorityParameters(11),
         new PeriodicParameters(
         new RelativeTime(0, 0),
         new RelativeTime(500, 0)),
         new StorageParameters(10000, null), 500);
      out = sps;
   }

   @Override
   public void handleAsyncEvent() {
      out.println("Ping␣" + cnt);
      ++cnt;
   }
}
```

Figure 4: The handler that writes periodically messages to the console.

added the mechanism of allocation in scoped memory to implement mission memory and private memory. It is possible to just use a single class, which we call `Memory`, to represent all three different SCJ memory types: immortal, mission, and private [19]. The `Memory` class is not intended to be used by application code, but RTSJ/SCJ memory classes are using it and delegating the scope handling to it. It should not be visible for application code. However, as the memory related classes are distributed between two different packages, it is practically visible.

4.1 Scheduler Simplification

The thread scheduler in JOP is implemented in Java. The thread scheduler in the JVM of JOP is invoked (1) on a timer interrupt, (2) when a thread finishes it's release (with the invocation of `waitForNextPeriod()`), and (3) when an event handler is fired. The scheduler performs three functions: (1) find the next thread to dispatch, (2) reprogram the timer interrupt, and (3) dispatch the thread. The thread with the highest priority, which is ready (it's release time is now or already passed), is selected for dispatch.

To find the time for the next timer interrupt, the priority ordered list of threads is searched. A thread shall only interrupt the currently dispatched thread, when its priority is higher than the dispatched thread. Therefore, only higher priority threads are searched. Within this set of threads, the thread with the *nearest* release time determines the next timer interrupt. That release time is used for the next scheduling interrupt.

The way the scheduler is implemented might not be the 'correct' way in which a real-time (RTSJ) scheduler shall be implemented. We use distinct (internal) priorities for all threads and not a FIFO queue for threads with the same priority. If two threads are assigned the same priority from the user, the scheduler implicitly provides a priority order. The reason for the current implementation is that it is optimized for low overhead. As the scheduler does not allow creating new threads during a mission, the data structures can be optimized at mission start.

```
final Runnable runner = new Runnable() {
  @Override
  public void run() {
    handleAsyncEvent();
  }
};

thread = new RtThread(priority.getPriority(), p, off) {
  public void run() {
      while (!MissionSequencer.terminationRequest) {
        privMem.enter(runner);
        waitForNextPeriod();
      }
  }
};
```

Figure 5: Reusing the JOP real-time threads to implement the SCJ periodic handler.

We do not want that a timer interrupt happens when no thread switch is needed. The scheduler is programmable timer based and not tick based. That is the reason why the timer is programmed for the next *higher priority* thread. We do not want to have an interrupt for a lower priority thread just to change the run queue and then switch back to the current thread.

The threads are collected in a simple array, which is priority ordered. There is no explicit run queue or other queues. However, the run queue is implicit: each thread that has a release time now or in the past is in the run queue. The first thread found in the priority ordered array is dispatched. This gives short dispatch time for high priority threads and longer for lower priority threads, which is ok as high priority threads have shorter deadlines when priorities are assigned deadline monotonic.

Threads check their deadlines and update their release time on a call to the `waitForNextPeriod()` method. The implementation of the SCJ handlers uses the original real-time threads from JOP (`RtThread`) [15]. Figure 5 shows the essential part of the SCJ `PeriodicEventHandler` implementation. Similar to a plain Java thread or RTSJ real-time thread, the `run()` method of `RtThread` is invoked when the thread is released. Within the periodic loop, on each release the initial private memory of the SCJ handler is entered with the `Runnable runner`. That `Runnable` in turn invokes, now in the private memory allocation context, `handleAsyncEvent()` of the SCJ style handler. For aperiodic events the `SwEvent` class of JOP's runtime is used.

4.2 Mission Startup and Termination

SCJ allows in all three levels mission sequences, with a mission memory that needs to be cleaned after each mission termination. As each mission may contain different number of threads (handlers), we expect that all data structures that are needed to represent threads in a JVM be allocated in mission memory. However, the problem is now how the scheduler shall find those thread objects. The scheduler is basically a JVM internal class/mechanism, which is allocated at boot time in immortal memory. According to the scope assignment rules, objects allocated in immortal memory are not allowed to point to objects in mission memory. To allow the scheduler to see the thread objects we need to violate this rule internally to the JVM. However, this is similar to other Java safety mechanism, such as implementing a GC in Java. In Java one cannot access memory directly, but if a GC is written in Java there needs to be some escape mechanism to access memory directly. The same is needed for the JVM internal scheduler to access thread objects in mission memory.

The scheduler in JOP is basically an interrupt handler, which is attached to the programmable timer interrupt at JVM startup. Another option is to allocate that interrupt handler object in mission memory and attach it to the interrupt on mission start and detach it at mission termination. However, at the moment too many applications still use the JOP `RtThread` classes for real-time threads. Therefore, we are not yet confident enough in our SCJ implementation to make such a fundamental change in the JOP runtime system.

4.3 Low-level Device Access

For low-level device access, the standard RTSJ style classes for direct memory access are used in SCJ. We would have preferred a more object-oriented way, the so-called hardware objects [21], to access low-level IO registers and DMA memory. For the implementation of the SCJ style low-level device access we can either use *native* system methods or hardware object based arrays. The system method version uses just static read and write methods, which access the memory mapped IO devices in JOP. With a hardware object based array, we can map the IO space into a Java array and then implement the raw memory classes by using that array.

4.4 Interrupts

SCJ (and the next version of RTSJ) has the notion of first-level interrupt handlers. An interrupt handler has to extend `InterruptServiceRoutine` and has to implement the interrupt handling code in the `handle()` method. The `InterruptServiceRoutine` object is used as lock to protect data structures that are used to communicate between the interrupt handler and the application threads or second level handlers. Interrupt priorities are higher than thread priorities. Executing a synchronized section with such an interrupt priority disables the scheduling interrupt and the interrupts at lower priority. The build tool or JVM runtime has to recognize this different handling of synchronized code for classes that extend `InterruptServiceRoutine`. As synchronized code blocks are not allowed in SCJ, only methods of such classes are involved. Therefore, it is well known at build time which methods need that special handling of locks with interrupt priority ceilings.

The notion of the interrupt handler in SCJ and the mapping of lock priorities to interrupt disabling is similar to the proposal in [20]. In our current implementation the JVM accepts a standard `Runnable` as an interrupt handler, which is registered by a system class. Even the thread scheduler is just a plain `Runnable` that is registered for the timer interrupt. Therefore, the SCJ implementation of interrupt handlers can keep the notion of the available `Runnable` and just invoke the `handle()` method of the interrupt handler. The hardware of JOP disables all interrupts, when an interrupt happens. By keeping them disabled during the `handle()` method and disabling them on all synchronized methods of the handler class we have a less responsive system, but are on the safe side for synchronization.

4.5 Packages and Java Friends

The SCJ related classes are defined within two packages (`javax.safetycritical` and `javax.realtime`). These classes need to share information for the implementation that shall not be visible at the application interface level. Member methods and fields can be protected at the package level. Therefore, building a library within one package is relative easy. Moving the whole SCJ related classes and APIs into a single package would simplify the implementation. However, even in that case on some systems it might be beneficial to collect low-level system functions (e.g., scheduling and memory management) into a system level class (in the JOP JVM those functions live in `com.jopdesign.sys`).

With C++ granting access to class internal methods and data to another, well known class is possible with the `friend` keyword, which is not available in Java. It is, however, somewhat possible to simulate this mechanism. One possibility is to create a singleton delegation class in the target package. The class has a private constructor, preventing application code from instantiating it. A singleton is created at class initialization. This object is then registered via a public static method in the class requiring access to the package, which only works once. Although this method is public, all subsequent invocations will simply be ignored.

4.6 Code Size

The JOP build tool links only classes used by the application into the final application. Furthermore, small versions of the JDK (a CLDC 1.1 version and a subset of it) are available for JOP. Therefore, the overhead of the JVM is relative small. The SCJ Hello World example, all needed SCJ and JDK support classes, and the JMV Java classes result in an application binary of 75 KB.

4.7 Source Access

The SCJ implementation on JOP is open source under the GNU GPL. The code is hosted on a `git` repository at GitHub and can be downloaded with

```
git clone git://github.com/jop-devel/jop.git
```

or as ZIP archive from

```
https://github.com/jop-devel/jop
```

Although the SCJ implementation is developed for JOP, it can be evaluated on a standard PC. The distribution contains a software simulation (`JopSim`) of the JOP JVM, written in Java. It is an interpreting JVM, which can read JOP application files and has the same restrictions as JOP. All JVM parts that are written in Java (e.g., garbage collection, thread scheduling) are interpreted with `JopSim` as it would be executed on the JOP hardware. Therefore, this simulation is a convenient tool for system level software debugging. `JopSim` can also be used for evaluation of SCJ. The *only* issue with a simulation that runs on a standard JVM on a non real-time operating system is that no timing can be guaranteed. For scheduling decisions the standard system time is used, which has usually a granularity of several milliseconds.

To run the JOP tool chain on a PC, Java and `make` need to be installed. The tools need to be built once with:

```
make tools
```

Building and running an application in the simulation is as follows:

```
make jsim -e USE_SCOPES=true P1=rtapi \
P2=examples/safetycritical P3=HelloSCJ
```

This example runs a SCJ hello world example where the `Mission` and the `Safelet` is combined into a single class. Furthermore, the single handler is instantiated as anonymous class. Therefore, this example fits into a single source file.

The meaning of the parameters is as follows: `USE_SCOPES`: JOP uses a garbage collector by default. This parameter enables scopes. `P1`, `P2`, `P3` describe the application:

- `P1` is the directory pointing to the Java source
- `P2` is the package name with '/' instead of '.' as delimiter
- `P3` is the main class

The SCJ hello world can be found in folder:
`jop/java/target/src/rtapi/examples/safetycritical`
Further information on the build process, and how to configure Eclipse for JOP development can be found in Chapter 2 of the JOP handbook [18].

The SCJ specification is still a moving target. So is the current implementation on JOP.

5. DISCUSSION

During the implementation of the SCJ on JOP we came across different aspects of the SCJ API specification we would like to change.

5.1 Asynchronous Event Handling

The notion of asynchronous events and asynchronous event handlers, which is inherited from the RTSJ, is probably overkill for SCJ and it does not work well for second level interrupt handlers. The current approach (at least in RTSJ 1.1) to fix this issue with a new `Happening` class makes the solution even more complex.

A simple solution to use SCJ style `AperiodicEventHandler` for software events and as a mechanism for second level interrupt handler would be:

1. Drop the inheritance from the RTSJ class `BoundAsyncEventHandler`
2. Drop the classes for aperiodic events (`AperiodicEvent` and `AsyncEvent`)
3. Add a method `release()` to the class `AperiodicEventHandler`

A software event is fired simply by invoking `release()` of the related aperiodic handler. This will release the underlying thread and it will be scheduled according to its priority. The same method can also be called from an interrupt handler to delegate some interrupt work to a second level handler.

There might be an argument that this solution is less flexible than the RTSJ style possibility to map several handlers to one event and/or several events to one handler. However, the same functionality can be implemented by the application when invoking several `release()` methods on a single event for the first case and invoke the `release()` from one handler at different events in the second case.[2]

[2]This suggestion of using a `release()` method on `AperiodicEventHandler` has been accepted by the SCJ expert group and is now part of the current specification. The `AperiodicEvent` and `AsyncEvent` have been removed from SCJ. The event handler still extends the RTSJ based event handler classes.

5.2 Application Startup

The SCJ defines a `Safelet` for the SCJ application and makes it vendor specific how the runtime knows which class will be the *main* class, which implements the `Safelet`. Our *vendor specific* solution is a standard Java `main()` method that needs to create the application class. The startup code for the SCJ example from Figure 1 is as follows:

```
public static void main(String[] args) {
  JopSystem.startMission(new HelloSafelet());
}
```

For a SCJ solution we need to define a static *start application* method in some SCJ class. This solution feels very natural for a Java programmer. The main argument against this Java `main()` method is that a SCJ application has to start with immortal memory and in standard Java (and in the RTSJ) `main` is executed in the heap memory context. However, if there is no GC the heap is implicitly immortal. A weaker argument is that the reference implementation (RI) shall run on top of the RTSJ and therefore this `main()` will run in heap. This can easily be solved by the same means as a `Safelet` is created and used in the RI. Use a different main for the execution of the RI, which first enters immortal memory and then invokes `main` of the SCJ application.

Using the common Java method to provide a static `main` method for SCJ application startup would also mean less vendor-specific Java code (or scripts) for an SCJ application.

As a compromise between the current vendor specific notion of the SCJ start and the `main()` method might be a static method with a different name and signature. It would make it clear that this application is not a standard Java application, but also avoid the vendor specific solutions for specifying and creation of the application object.

5.3 Immortal Memory

It is convenient to allocate some data structures, e.g., a reference to the console, shared data for a single mission, or shared data between missions, in immortal memory. References to such data can then be stored in static fields and are therefore easily accessible by event handlers. However, no user visible method is executed in immortal memory. Two possibilities exist to allocate shared data in immortal memory: (1) enter immortal memory from mission initialization or (2) use static class initializers, which are executed in immortal memory.

Entering immortal memory from the first mission initialization method is cumbersome and also ties allocation in immortal memory to a dedicated mission. Using static class initializers is fine for short code sequences, but becomes also cumbersome when more initialization code is needed. Furthermore, extensive usage of static class initializers can easily lead to cyclic dependencies of static class initializers, which is disallowed in SCJ.

An easy solution would be a method on `Safelet` that is executed by the SCJ infrastructure in the immortal memory allocation context before a mission sequencer is requested and mission memory objects are created. This method could be called `initialize()`, similar to the initialization method of a mission, which is executed in mission memory.

6. CONCLUSIONS

The definition of a standard specification needs an implementation of it and usage applications to evaluate if the specification is sound. This is even important before the specification is finished. We have implemented a prototype version of the upcoming safety-critical Java standard on the Java processor JOP. We did not run into any main issues that hamper the implementation of SCJ. However, as the specification is still under development, we need to continuously update our implementation of it.

7. ACKNOWLEDGMENTS

The Authors would like to thank Torur Strom for the suggestion of the singleton delegation mechanism to simulate friend classes in Java. Furthermore, we would like to thank the reviewers for their long and detailed reviews that helped to improve the paper.

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

8. REFERENCES

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

[2] Austin Armbruster, Jason Baker, Antonio Cunei, Chapman Flack, David Holmes, Filip Pizlo, Edward Pla, Marek Prochazka, and Jan Vitek. A real-time Java virtual machine with applications in avionics. *Trans. on Embedded Computing Sys.*, 7(1):1–49, 2007.

[3] Thomas Bøgholm, René R. Hansen, Anders P. Ravn, Bent Thomsen, and Hans 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.

[4] Greg Bollella, James Gosling, Benjamin Brosgol, Peter Dibble, Steve Furr, and Mark Turnbull. *The Real-Time Specification for Java*. Java Series. Addison-Wesley, June 2000.

[5] Alan Burns, Brian 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.

[6] Thomas Henties, James J. Hunt, Doug Locke, Kelvin Nilsen, Martin Schoeberl, and Jan 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.

[7] Jagun Kwon, Andy Wellings, and Steve 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.

[8] Doug Locke, B. Scott Andersen, Ben Brosgol, Mike Fulton, Thomas Henties, James J. Hunt, Johan Olmütz Nielsen, Kelvin Nilsen, Martin Schoeberl, Joyce Tokar, Jan Vitek, and Andy Wellings. Safety-critical Java technology specification, public draft, 2011.

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

[10] Kelvin 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.

[11] Filip Pizlo, Lukasz Ziarek, and Jan Vitek. Real time Java on resource-constrained platforms with Fiji VM. In *Proceedings of the 7th International Workshop on Java Technologies for Real-Time and Embedded Systems (JTRES 2009)*, pages 110–119, New York, NY, USA, 2009. ACM.

[12] Ales Plsek, Lei Zhao, Veysel H. Sahin, Daniel Tang, Tomas Kalibera, and Jan 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.

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

[14] Juan Ricardo Rios and Martin 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.

[15] Martin Schoeberl. Real-time scheduling on a Java processor. In *Proceedings of the 10th International Conference on Real-Time and Embedded Computing Systems and Applications (RTCSA 2004)*, Gothenburg, Sweden, August 2004.

[16] Martin 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.

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

[18] Martin Schoeberl. *JOP Reference Handbook: Building Embedded Systems with a Java Processor*. Number ISBN 978-1438239699. CreateSpace, August 2009. Available at `http://www.jopdesign.com/doc/handbook.pdf`.

[19] Martin 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.

[20] Martin Schoeberl, Stephan Korsholm, Tomas Kalibera, and Anders P. Ravn. A hardware abstraction layer in Java. *ACM Trans. Embed. Comput. Syst.*, 10(4):42:1–42:40, November 2011.

[21] Martin Schoeberl, Stephan Korsholm, Christian Thalinger, and Anders 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.

[22] Martin Schoeberl, Hans Sondergaard, Bent Thomsen, and Anders 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.

[23] Neeraj Kumar Singh, Andy Wellings, and Ana Cavalcanti. The cardiac pacemaker case study and its implementation in safety-critical Java and Ravenscar Ada. In *Proceedings of the 10th International Workshop on Java Technologies for Real-Time and Embedded Systems (JTRES 2012)*, Copenhagen, DK, October 2012. ACM.

[24] Hans Søndergaard, Stephan E. Korsholm, and Anders 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 2012)*, Copenhagen, DK, October 2012. ACM.

[25] Tórur Biskopstø Strøm and Martin 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)*, Copenhagen, DK, October 2012. ACM.

[26] Andy Wellings and Martin 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.