Virtualizing Real-time Embedded Systems with Java

Jan Vitek
Computer Science Department
Purdue University

ABSTRACT

Real-time embedded systems come in all shapes and sizes with vastly different capabilities. They often operate under stringent resource constraints, ranging from space and time to power. Programming them is usually done in low-level system's programming languages close to the hardware. The resulting software is costly and not particularly portable. The Java programming language has been successful in providing a virtualized, high-level, development environment for desktop and server applications. Programming in Java leads to memory-safe code that can be ported straightforwardly across architecture and operating system. This paper surveys the state of the art in Java virtualization for real-time and embedded systems. Technological advances in virtual machines as well as new real-time extensions to the language have brought Java closer to being widely usable for a wide range of embedded problems.

1. INTRODUCTION

Real-time embedded systems must operate in the presence of strong resource constraints with widely different hardware, operating system, and programming style. Yet there are important commonalities. Performance and predictability of the software are usually paramount. Software is written to be verifiable with simple control flow and little memory allocation. The programming languages of choice are subsets of C or Ada, for the control they afford over the underlying resources. Unfortunately, low-level languages often entail software faults and lack of portability of the resulting code. Both of which are becoming issues as the size of real-time code bases increases. Factors such as productivity, reusability, and availability of trained personnel have spurred interest in Java as an alternative. Scaling a high-level language and its runtime libraries down to resource-constrained embedded settings is difficult, especially when hard real-time guarantees must be met. Virtual machines are usually tuned and engineered for maximal throughput. Implementation techniques include fast paths for the common case, and expensive slow-paths for less frequent cases (e.g. inline caches on interface dispatch and type tests). Support for real-time requires an implementation that targets predictability over throughput.

The benefits of Java over low-level systems programming languages can be summarized by the following (partial) list:

- **Object-oriented Programming:** Java relies on object-oriented principles to foster software reuse. Concepts like classes, interfaces and late binding facilitate the design of reconfigurable systems.
- **Concurrency:** Java has builtin concurrency primitives for mutual exclusion and threading, as well as thread-safe libraries and a memory model that constrains the optimization compilers can perform. These are features lacking from languages such as C where concurrency is an afterthought and compilers are likely to generate unsafe code.
- **Memory Safety:** The language provides a high-level memory API. Strong type safety is enforced by the compiler. Array bounds and type casts are checked at run-time, and explicit frees are not allowed. This ensures that an invalid memory access will be trapped and lead to a proper runtime exception.
- **Dynamic memory management:** Java mandates that unused memory be reclaimed by a garbage collection algorithm. This simplifies development of concurrent code as the protocols for freeing memory in the presence of multiple threads are tricky.
- **Dynamic code loading:** Java support the loading of class files at runtime, thus permitting programmers to extend or patch deployed applications.
- **Rich libraries:** The Java development environment comes with an extensive set of well-tested and documented libraries ranging from graphic interfaces to natural language processing.

One last non-technical advantage of Java is the large number of trained programmers. For a long time these "advantages" were considered as liabilities in the embedded and real-time community. Object-oriented programming and memory safety were sources of run-time overheads. Concurrency was not as relevant when all the embedded devices were uniprocessors. Dynamic memory management and dynamic loading were source of timing unpredictability. The libraries were too large and did not meet the requirement of resource constrained devices. Lastly, there was no support for writing code that had to meet strict timing constraints.

The tide has begun to turn. A number of trends are conspiring to make Java virtual machines the platform of choice for real-time embedded development: Embedded hardware is becoming more powerful; Multi-core usage is trending upwards; The functionality implemented in software is growing along with the size of the code bases; Security and safety requirements are increasingly a concern. This paper provides a short survey of the state of the art in real-time embedded development in Java focusing on key advances that are expending the applicability of the technology and using the author's experience on several real-time Java virtual machine to illustrate the promises of Java in this field.

2. VIRTUALIZING REAL-TIME

The key to virtualization is to devise an API that hides the underlying infrastructure without significant loss of expressiveness or performance. Real-time systems are particularly dependent on the timing facilities provided the hardware, the operating systems for synchronization, timers and scheduling, among other. These are the things that need to be virtualized. There are two efforts under way to virtualize the hardware and the operating system. The first is the Real-time Specification for Java (RTSJ) [6] and more recently the Safety Critical Java Specification (SCJ) [12]. While both provide an API for real-time programming on the Java virtual machine, the SCJ is more forceful about ensuring predictability and verifiability.

The Real-Time Specification for Java (RTSJ) had the goal to "provide an Application Programming Interface that will enable the creation, verification, analysis, execution, and management of Java threads whose correctness conditions include timeliness constraints" [6] through a combination of additional class libraries, strengthened constraints on the behavior of the JVM, and additional semantics for some language features, but without requiring special source code compilation tools. The RTSJ covers five main areas related to real-time programming.

- *Scheduling*: Priority based scheduling guarantees that the highest-priority schedulable object is always the one that is running. The scheduler must also support the periodic release of real-time threads, and the sporadic release of asynchronous event handlers that can be attached to asynchronous event objects that themselves are triggered by actual events in the execution environment.

- *Admission control and cost enforcement*: Schedulable objects can be assigned parameter objects that characterize their temporal requirements in terms of start times, deadlines, periods, and cost. This information can be used to prevent the admission of a schedulable object if the resulting system would not be feasible.

- *Synchronization*: Priority inversion through the use of Java's synchronization mechanism are controlled by using the priority inheritance protocol, or optionally, the priority ceiling emulation protocol.

- *Memory Management*: Time-critical threads must not be subject to delays caused by garbage collection. To facilitate this, a no-heap real-time thread is prohibited from touching heap allocated objects, and so can preempt garbage collection at any time. Instead of using heap memory, these threads can use special, limited-lifetime memory areas known as *scoped memory areas.*

- *Asynchronous Transfer of Control*: It is sometimes desirable to terminate a computation at an arbitrary point. The RTSJ allows for the asynchronous interruption of methods [7]. This facilitates early termination while preserving the safety of code that does not expect such interruptions.

The complexity of safety critical software varies between application; the SCJ specification gives three virtualized APIs called compliance levels. The first, Level 0, provides a simple, frame-based cyclic executive model which is single threaded with a single mission. Level 1 extends this model with multi-threading via aperiodic event handlers, multiple missions, and a fixed-priority preemptive scheduler. Level 2 lifts restrictions on threads and supports nested missions.

SCJ programs are organized as *missions*, which are independent computational units with respect to lifetime and resources needed. Each mission is composed of a bounded number of *schedulable objects*. Missions are launched according to a pre-defined order. Figure 1 shows the three phases of a mission: *initialization*, *execution*, and *cleanup*. After a mission terminates, the next mission is released if there is one.

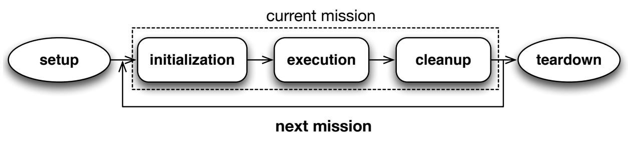

Figure 1: Mission Life Cycle.

Schedulable objects contains both computation logic and some scheduling constraints, such as release time, deadline, priority, and so on. In the SCJ, schedulable objects have a dedicated thread and are restricted to periodic event handlers (PEH), aperiodic event handlers (AEH), and managed threads (MT) to simplify feasibility analysis.

At Level 0, the *cyclic executive model* defines a mission as a set of computations, each of which is executed periodically in a precise, clock-driven timeline, and processed repetitively throughout the mission. The only schedulable objects permitted at Level 0 are Periodic Event Handlers. All PEHs execute under control of a single underlying thread, so the implementation can safely ignore synchronization in the application. In this scenario, an operation which blocks will block the entire application.

Figure 2 illustrates some of the core classes of the SCJ. At The primordial thread starts in immortal memory and creates a mission sequencer after executing the setup procedure. The mission sequencer holds references to all of the missions and repeatedly selects the next mission to launch. Upon launching a mission, the mission memory is allocated with the desired size. A mission manager is then created, in mission memory, to control the mission's schedulable objects. The three phases of the mission are all executed in mission memory. All schedulable objects are created in the initialization phase of the mission; they are then started upon entering execution phase. A mission runs forever unless a termination request is sent explicitly by one of its

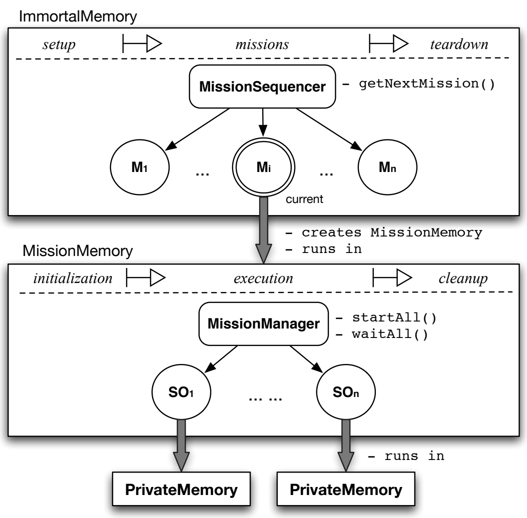

Figure 2: SCJ Mission classes.

schedulable objects. If a termination request is sent, the mission enters the cleanup phase. The cleanup phase includes storing or reporting the outcomes of a mission, as well as releasing all acquired resources. Once the mission is completed, all mission-specific objects are deallocated as the mission memory is reclaimed.

The scope-based memory model introduced by RTSJ is retained in the SCJ specification. The main differences with the RTSJ are that the heap has been completely abandoned and that scoped memory has been further restricted to make certification easier. There are three types of memory areas: *immortal memory*, *mission memory*, and *private memory*. Immortal memory spans the lifetime of the virtual machine; therefore, only objects that should survive the entire program execution should be allocated there. The lifetime of the latter two memory areas is bounded. Each mission has a mission memory which is shared by the mission's schedulable objects and used to allocate data that must persist throughout the mission. Each schedulable has its own private memory area for the data that is needed for only a single activation of the schedulable. The `enterPrivateMemory(s, logic)` method creates a nested private memory of size `s`, enters it, and execute the `logic` within it.

A scope stack is a logical data structure that represent the scoped memory areas that a given schedulable object has entered. Figure 3 illustrates this with the stacks of schedulable objects — two periodic event handlers. The stacks grow logically from immortal memory up to private memories. Unlike the RTSJ where cactus stacks are allowed, the SCJ specification restricts navigation to linear sequences of

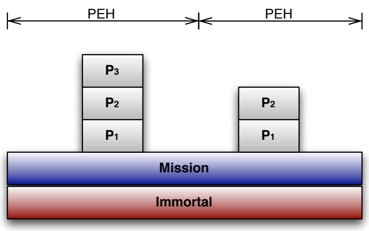

Figure 3: Memory structure of a SCJ program.

scopes. This is achieved by removing the RTSJ's `enter()` method and replacing it with the more restricted `enterPrivateMemory()` which which only creates a subscope if there isn't one active already.

For memory safety, every write to a field holding an object reference must be checked to prevent dangling references. An optional set of annotations can be used to prove that at compile-time that all stores are safe. When these annotations are used, an implementation is permitted to omit scope checks.

3. CONCRETE IMPLEMENTATIONS

Implementations of Java for embedded devices come in three rough (and sometimes overlapping) classes:

- **Interpreters:** Typically deal with Java bytecode directly, interpreter are slow, but reasonably memory efficient and supports dynamic class loading.
- **Just-in-time compilers:** JITs generate code on the fly and vary in performance depending on the level of optimization applied to the bytecode. They are typically more resource hungry and can introduce pauses in the execution of the system.
- **Ahead-of-time compilers:** AOTs can generate highly optimized code by compilation to either C or directly to native code. They usually do not support dynamic class loading (which allowed in SCJ).

RT JVMs must choose a memory management policy:

- **RTGC:** Some JVM choose to allow standard Java semantics and rely on a real-time garbage collector to bound the pauses due to memory management.
- **Scopes + GC:** The RTSJ mandates scoped memory (which is not GCed) as well as a heap that is collected by a (possibly RT) GC.
- **Scopes:** The SCJ does not support automatic memory management, all allocation is in scope memory regions.

Four commercial JVMs support the RTSJ. These are WebSphere Real-time [2], Java RTS [5], PERC [19], and Jamaica [21]. PERC is noteworthy in that it supports features similar to those offered by the RTSJ but uses its own APIs. All of these systems support real-time garbage collection, though the algorithms are markedly different ranging from time-based to work-based with varying degrees of support for concurrency. In addition, Oracle's WebLogic Real-Time and Azul Systems' virtual machine [9] both offer low-pause-time garbage collectors that approach real-time performance. The execution strategies of these systems range from ahead-of-time compilation to just-in-time compilation. PERC and Jamaica are the only other products currently targeting resource constrained embedded devices.

Ovm [1] is a Java-in-Java metacircular virtual machine that provides hard real-time guarantees. It generates C code. It was used in the first Unmanned Aerial Vehicle flight using avionics software written in Java [1] and the first *open-source* real-time garbage collector with high throughput and good mutator utilization [16]. Ovm lacks functionality essential for embedded hard real-time systems. Many applications run on minimal hardware with a small real-time OS kernel and performance close to C. Ovm's performance and footprint prevented us from experimenting with a wide range of embedded devices. Furthermore, Ovm suffered from an overly complex design and we could not envision how the system could ever be certified. JRate was a contemporary of Ovm that was integrated into the GCC compiler [10].

Real-time garbage collection has been investigated for many years in the context of Java. Nielsen [15], Baker's [4] and Henriksson's [11] early works inspired a number of practical algorithms including the ones used in all commercial VMs. The IBM Metronome collector uses periodic scheduling [3] to ensure predictable pause times. Jamaica uses a work-based techniques with fragmented objects [20] to get around the need for a moving collector to fight fragmentation. Java RTS [8] uses a non-moving variant of Henriksson with a scheduler that leverage the slack in real-time systems to collect garbage when no real-time task is active. For overviews of the main alternatives readers are referred to [14, 17].

4. REAL-TIME JAVA IN ACTION

Most of the SCJ functionality can be implemented in the library with a clear interface to the underlying virtual machine to ease the task of porting it across VMs. For the virtual machine, we have modified Ovm [1]. The runtime support for our VM is lightweight. It contains two components: the memory manager and an OS abstraction layer for thread management. The runtime currently runs on POSIX-like platforms such as Linux, NetBSD, and on top of the RTEMS classic API.

We compare Java to hand-written C code in performance and predictability. To this end, we set up a representative workload on a realistic platform. For our workload, we selected the CDx [13] benchmark. The benchmark consists of a periodic task that takes air traffic radar frames as input and predicts potential collisions. The benchmark itself measures the time between releases of the periodic task, as well as the time taken to compute potential collisions. For our evaluation, we used a pre-simulated formula to generate the radar frames to achieve a consistent workload across executions. The C implementation [18] matches closely the Java version, with one source file per Java class. Our experiments were run on a GR-XC3S-1500 LEON development board. The board's Xilinx Spartan3-1500 field programmable gate array was flashed with a LEON3 configuration, without a floating-point unit, running at 40MHz, with an 8MB flash PROM and 64MB of PC133 SDRAM split into two 32MB banks. The version of RTEMS is 4.9.3. Our second platform is an Intel Pentium 4 3.80GHz single core machine with 3GB of RAM, running Ubuntu Linux 9.04 with the 2.6.28-15-generic 32-bit SMP kernel.

SCJ vs. C on LEON3. In our first experiment, we compared C with SCJ on the LEON3 platform. The periodic task runs every 120 milliseconds with 6 airplanes and 10,000 iterations. Figure 4 gives the runtime performance. No deadlines were missed in any executions. On average, the execution time of one iteration in C is around 53 milliseconds, while for Java it is around 69 milliseconds. The median execution times for C are only 28% smaller than the median for Java. For real-time developers, the key metric of performance is the worst case. C is 34% faster than SCJ in the worst-case. Figure 7 shows a more detailed view of a subset of the iterations. There is a strong correlation of execution times between C and Java. In this benchmark, Java is as predictable as C.

SCJ vs. C on x86. On the x86 we configured the benchmark to run with a more intensive workload, using 60 planes with a 60 millisecond period for 10,000 iterations. The increased number of planes brings more detected collisions, which consequently poses higher demands on data structures, arithmetics, and memory allocation. The histogram in Figure 6 shows the frequency of execution times for SCJ and C. The data demonstrates that on average Java is by 12% faster than C. Looking at the worst-case performance times, Java has a 4% overhead over C. We can observe again that the results are highly correlated. In fact, for the most of the time, Java performance times stay below those of C It is unclear why Java would be faster than C. We believe that Ovm facilitates inlining by generating a single C file and removing most of the polymorphism around methods calls, but this should have the same effect on both platform. The x86 workload performs more allocation and C uses `malloc/free` which is more expensive than the bump-pointer allocation used for scoped memory in SCJ.

Program Example. Figure 8 shows two key classes of the benchmark: `Level0Safelet` and `CollisionDetectorHandler`. The `Level0Safelet` class extends the `CyclicExecutive` class and represents an instance of a Level 0 mission. It defines the specific actions that will be executed during the initialization (the `setUp()` method) and the cleanup of the mission (the `cleanUp()` method). First, the SCJ infrastructure asks for required mission space by calling `missionMemorySize()` method and creates a corresponding mission area. Furthermore, the `initialize()` method is executed to instantiate all the periodic event handlers that will be periodically executed during the mission. Therefore, the infrastructure instantiates the `CollisionDetectorHandler` class and creates a corresponding private memory - the size of the private memory is given in the constructor of the handler. Finally, the `getSchedule()` represents a Level 0 scheduler and defines the frequency of the handler execution. The `CollisionDetectorHandler` class represents a `PeriodicEventHandler` dedicated to the mission and defines the code that will be executed each time the handler is scheduled to run. The handler is executed by the infrastructure code calling the `handleEvent()` method. Inside the `handleEvent()` method we delegate the functionality to the `runDetectorInScope()` method that first receives a data frame holding the current positions of the aircraft. Finally, note that both the classes contain the SCJ annotations specifying their level and runtime allocation context, these annotations are used during the static analysis to prove allocation safety of the application.

5. CONCLUSION

This paper has present a succinct overview of the state of the art in real-time programming on the Java platform. We have illustrated the ease of developing real-time application and the low overheads of modern implementations of the languages.

Acknowledgments This work was partially supported by NSF grants CNS-0938256, CCF-0938255, CCF-0916310 and CCF-0916350. The author thank Tomas Kalibera, Petr Maj, Ales Plsek, Filip Pizlo Veysel H. Sahin, Daniel Tang, and Lei Zhao.

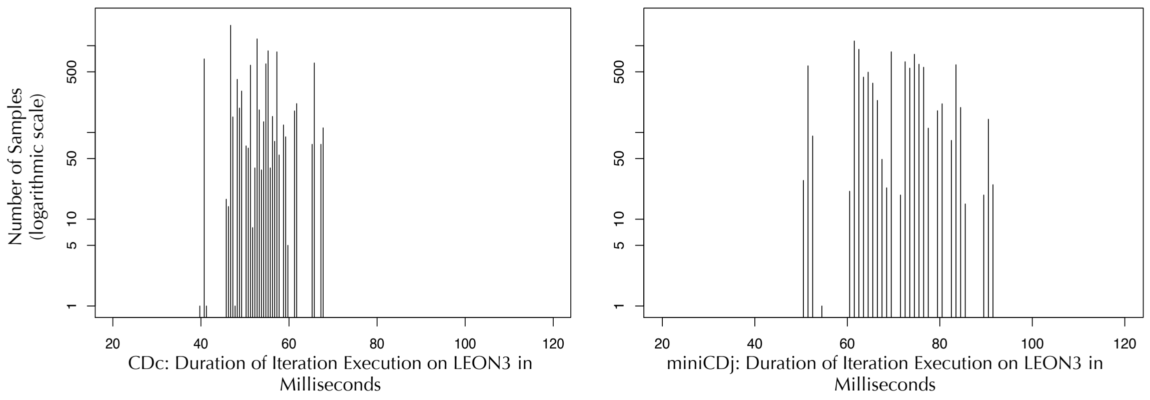

Figure 4: Histograms of execution times on LEON3. The observed worst-case for C is 34% faster Java.

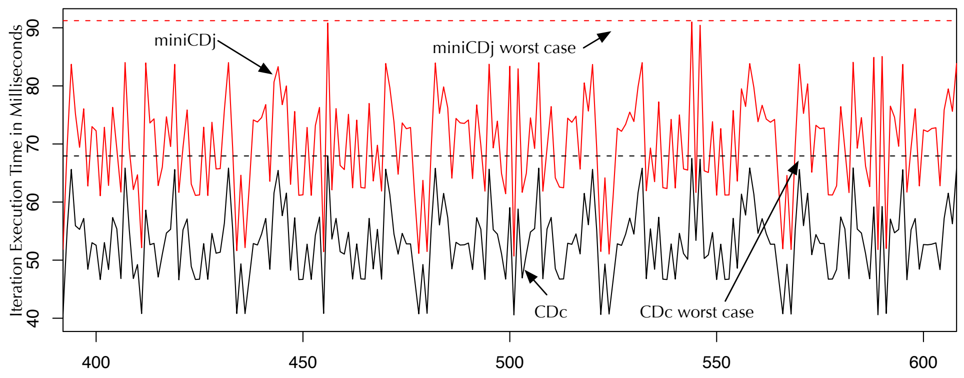

Figure 5: A detailed runtime comparison of for 200 iterations on LEON3.

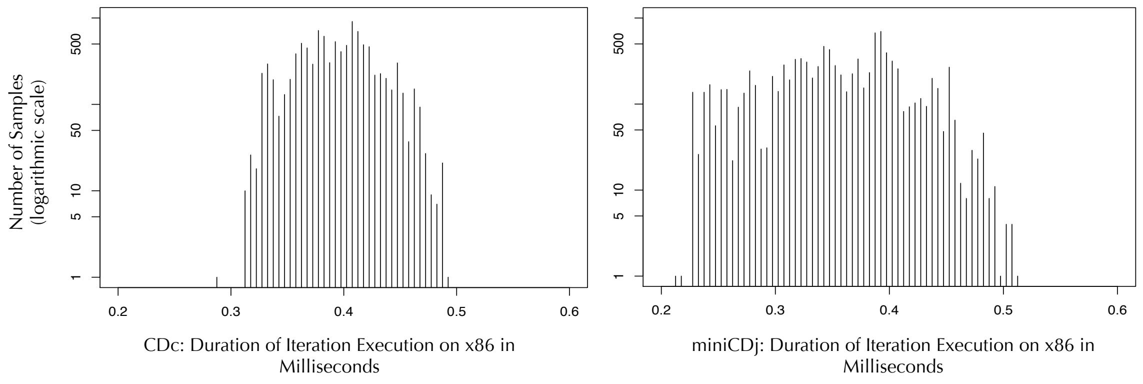

Figure 6: Histograms of execution times on X86. The observed worst case for C is 4% faster than Java.

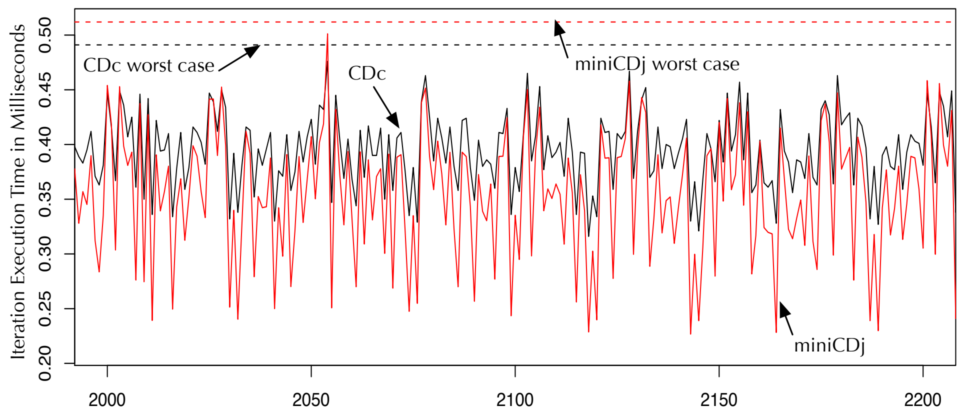

Figure 7: A detailed runtime comparison of C and Java for 200 iterations on x86.

```
@SCJAllowed(value=LEVEL_0, members=true)
@Scope("immortal")
public class Level0Safelet extends CyclicExecutive {
  public Level0Safelet() { super(null); }

  public void setUp() {
    new ImmortalEntry().initialize();
    new Simulator().generate();
  }

  @RunsIn("cdx.Level0Safelet")
  protected void initialize() {
    new CollisionDetectorHandler();
  }

  public void tearDown() { dumpResults(); }

  public CyclicSchedule getSchedule(
                        PeriodicEventHandler[] handlers) {
    CyclicSchedule.Frame[] frames = new CyclicSchedule.Frame[1];
    frames[0] = new CyclicSchedule.Frame(
                  new RelativeTime(PERIOD, 0), handlers);
    return new CyclicSchedule(frames);
  }

  public long missionMemorySize() { return DETECTOR_SIZE; }
}
```

```
@SCJAllowed(value=LEVEL_0, members=true)
@Scope("cdx.Level0Safelet")
@RunsIn("cdx.CollisionDetectorHandler")
public class CollisionDetectorHandler
                  extends PeriodicEventHandler {

  private final TransientScopeEntry cd = new
    TransientScopeEntry(new StateTable(), VOXEL_SIZE);
  public boolean stop;

  public CollisionDetectorHandler() {
    super(null, null, null, TRANSIENT_SIZE);
  }

  public void handleEvent() {
    if (stop)
      Mission.getCurrentMission().requestSequenceTermination();
    else
      runDetectorInScope(cd);
  }

  public void runDetectorInScope(TransientScopeEntry cd) {
    RawFrame f = ImmortalEntry.frameBuffer.getFrame();
    cd.setFrame(f);
    cd.processFrame();
    ImmortalEntry.frames++;
     stop = (ImmortalEntry.frames == MAX_FRAMES);
  }
}
```

Figure 8: Code example showing `Level0Safelet` and `CollisionDetectorHandler` classes.

6. REFERENCES

[1] A. Armbuster, J. Baker, A. Cunei, D. Holmes, C. Flack, F. Pizlo, E. Pla, M. Prochazka, and J. Vitek. A Real-time Java virtual machine with applications in avionics. *Transactions in Embedded Computing Systems*, 7(1):1–49, 2007.

[2] J. Auerbach, D. Bacon, B. Blainey, P. Cheng, M. Dawson, M. Fulton, D. Grove, D. Hart, and M. Stoodley. Design and implementation of a comprehensive real-time Java virtual machine. In *Conference on Embedded software*, 2007.

[3] D. Bacon, P. Cheng, and V. Rajan. A real-time garbage collecor with low overhead and consistent utilization. In *Symposium on Principles of Programming Languages*, 2003.

[4] H. G. Baker. The treadmill: real-time garbage collection without motion sickness. *SIGPLAN Notices*, 27(3):66–70, 1992.

[5] G. Bollella, B. Delsart, R. Guider, C. Lizzi, and F. Parain. Mackinac: Making hotspot real-time. In *Symposium on Object-Oriented Real-Time Distributed Computing*, 2005.

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

[7] B. Brosgol, S. Robbins, and R. Hassan. Asynchronous transfer of control in the RTSJ. In *Symposium on Object-Oriented RT Distributed Computing*, 2002.

[8] E. Bruno and G. Bollella. *Real-Time Java Programming: With Java RTS*. Addison-Wesley, 2009.

[9] C. Click, G. Tene, and M. Wolf. The pauseless GC algorithm. In *Conference on Virtual Execution Environments (VEE)*, 2005.

[10] A. Corsaro and D. Schmidt. The design and performace of the jRate Real-Time Java implementation. In *Symposium on Distributed Objects and Applications*, 2002.

[11] R. Henriksson. *Scheduling Garbage Collection in Embedded Systems*. PhD thesis, Lund University, 1998.

[12] T. Henties, J. Hunt, D. Locke, K. Nilsen, M. Schoeberl, and J. Vitek. Java for safety-critical applications. In *Certification of Safety-Critical Software Controlled Systems (SafeCert)*, 2009.

[13] T. Kalibera, J. Hagelberg, F. Pizlo, A. Plsek, Ben Titzer and J. Vitek. Cdx: A family of real-time Java benchmarks. In *l Workshop on Java Technologies for Real-time and Embedded Systems*, 2009.

[14] T. Kalibera, F. Pizlo, A. Hosking, and J. Vitek. Scheduling hard real-time garbage collection. In *Real-Time Systems Symposium*, 2009.

[15] K. Nilsen. Garbage collection of strings and linked data structured in real time. *Software, Practice & Experience*, 18(7):613–640, 1988.

[16] F. Pizlo and J. Vitek. An empirical evalutation of memory management alternatives for Real-time Java. In *Real-Time Systems Symposium*, 2006.

[17] F. Pizlo and J. Vitek. Memory management for real-time java. In *Symposium on Object-oriented RT Distributed Computing*, 2008.

[18] F. Pizlo, L. Ziarek, E. Blanton, P. Maj, and J. Vitek. High-level programming of embedded hard real-time devices. In *EuroSys Conference*, 2010.

[19] T. Schoofs, E. Jenn, S. Leriche, K. Nilsen, L. Gauthier, and M. Richard-Foy. Use of PERC pico in the AIDA avionics platform. In *Workshop on Java Technologies for Real-Time and Embedded Systems*, 2009.

[20] F. Siebert. The impact of realtime garbage collection on realtime Java programming. In *Symposium on Object-Oriented RT Distributed Computing*, 2004.

[21] F. Siebert and A. Walter. Deterministic execution of Java's primitive bytecode operations. In *Java Virtual Machine Research and Technology Symposium*, 2001.