Safety Critical Java for Robotics Programming

Bent Thomsen*, Kasper Søe Luckow*[1], Thomas Bøgholm*, Lone Leth Thomsen*,
Stephan Erbs Korsholm†
*Department of Computer Science, Aalborg University
{bt,luckow,boegholm,lone}@cs.aau.dk
†VIA University College
sek@viauc.dk

Abstract—**This paper introduces Safety Critical Java (SCJ) and argues its readiness for robotics programming. We give an overview of the work done at Aalborg University and elsewhere on SCJl, some of its implementations in the form of the JOP, FijiVM and HVM and some of the tools, especially WCA, TetaSARTS tool suite and SymRT, allowing programmers to analyze their SCJ applications for correct time behaviour.**

Since its inception in 1995, Java has become one of the most popular programming languages. With its *write once - run anywhere* approach, it was rapidly propelled into mainstream computing. The robotics community quickly started to appreciate the language, and today Java is very popular in robotics programming with several academic and commercial frameworks [48], [50], and a very active community [21].

It is well known that Java, in its standard edition, is unsuited for hard real-time applications, mainly due to insufficient thread semantics and the use of unpredictable garbage collection algorithms in the implementation of the Java Virtual Machine (JVM). Clearly some areas of robotics have hard real-time requirements. Here ad-hoc approaches or approaches based on external code, often written in C running on real-time operating systems such as RT_PREEMPT and accessed via the Java Native Interface (JNI), have prevailed. Although impressive systems, such as the programming of Humanoid Robots [57], have been implemented this way, it requires the programmer to juggle two programming languages and run-time environments with very different and sometimes conflicting semantics.

To accommodate hard real-time applications in Java, much research has been devoted to developing appropriate programming models in Java facilitating real-time systems development. The first such approach was conducted under the very first Java Community Process (JSR001) and led to the Real-Time Specification for Java (RTSJ) [12]. RTSJ introduces high resolution clocks, the notion of *NoHeapRealTimeThread* and scoped memory, allowing application developers to program threads that have no interaction with the heap to avoid problems with the garbage collector during real-time execution. As reported in [58] RTSJ has been used in the M2V2 humanoid robot [46] and there are several examples of RTSJ being used for industrial robotics [60], [49].

However, RTSJ is rather resource demanding and the original implementation from SUN required a high end Sparc processor running a Solaris 10 operating system, though newer versions from Oracle, as well as commercial versions of RTSJ from other vendors, have reduced the resource demands considerably [1], [3]. RTSJ is fairly dynamic in nature, leaving checks of consistent use of scoped memory and real-time properties to run-time analysis. Thus RTSJ is not suitable for static verification of hard real-time properties, and in many cases it is not suitable for small embedded systems. Therefore Java has so far been absent in the area of robotics on small embedded platforms used for hard real-time systems requiring certification of time properties.

In recent years the Java community, through JSR 302, has made tremendous progress, and an important step has been taken with the upcoming Safety Critical Java (SCJ) standard [36] making Java a viable choice for development of embedded hard real-time systems. The SCJ standard is an extended subset of RTSJ; it similarly contains high-resolution clocks and timers, the idea of *NoHeapRealTimeThread* and (a simplified version of) the scoped memory model to remove the need for a garbage collector. In addition, SCJ has a sufficiently tight thread semantics and a programming model based on tasks grouped in missions, contributing to facilitating the task of verifying real-time properties. A mission encapsulates a specific functionality or phase in the lifetime of the real-time system as a set of schedulable entities. For instance, a flight-control system may be composed of take-off, cruising, and landing, each of which can be assigned a dedicated mission. A schedulable entity handles a specific functionality and has release parameters describing the release pattern and temporal scope in terms of release time, deadline, etc. The release pattern is either periodic or aperiodic following a classic control system structure [14].

There are now a number of JVM implementations supporting the SCJ programming model including FijiVM [43], The Hardware-near Virtual Machine (HVM) [59], [32], [38] and the Java Optimized Processor (JOP) [51]. FijiVM has sufficiently low memory demands and is applicable for embedded systems. It does, however, require a POSIX-like OS. HVM is a lean JVM implementation directed towards use in resource-constrained embedded devices with as low as 256 KB ROM and 20 KB RAM. It features both iterative interpretation, Java-to-C compilation, and a hybrid of the two [32]. The HVM is self-contained and does not rely on an OS. It runs on popular robotics platforms such as Atmel AVR ATmega2560 microcontroller, Arduino and Lego EV3. The JOP is a JVM implemented in hardware using an FPGA. Both JOP and the HVM support the notion of Hardware Objects [33], an object-

[1]Now at Carnegie Mellon University, Silicon Valley, USA.

oriented abstraction of low-level hardware devices, such as I/O registers and interrupts, which can be handled directly from Java space.

Rigorous verification is essential for safety critical embedded hard real-time systems needing to comply with tight timing constraints, especially for systems needing to comply with standards such as DO-178C, ISO-26262, IEC-61508 and EN-50128 (applying to systems operating in safety-critical domains such as avionics and automotive). Of special interest is the verification of the system being *schedulable* i.e. verifying that all real-time tasks, under the conditions of the employed scheduling policy, finish before their respective deadlines in all circumstances. Response time analysis [14] is a traditional approach for concluding on schedulability; the response times of the real-time tasks are calculated using Worst Case Execution Time (WCET) and blocking times, and the system is schedulable if the response times are less than the task deadlines. Traditionally WCET analysis has been done by measurement, which provides an unsafe estimation approach. In recent years, a number of tools for timing analyses of systems written in Java have emerged; WCA [54] for WCET analysis on JOP, TetaJ [27] for WCET analysis on HVM, SARTS [11] for schedulability analysis on JOP, and TETASARTS [41] for schedulability and other time analyses on JOP and HVM. All the mentioned tools follow a strategy of translating SCJ programs into networks of Timed Automata, and timing analysis is then formulated in terms of an appropriate logic, e.g. Timed Computation Tree Logic (TCTL), and subjected to model checking using UPPAAL [5]. The SARTS and TETASARTS tools take the interactions between tasks into account during schedulability analysis, therefore systems deemed unschedulable using traditional response time analysis may be deemed schedulable using model checking as demonstrated in [11]. TETASARTS allows the programmer to write the program in a platform independent way, using the SCJ profile, and then analyse whether it can be scheduled on the particular platform. Hence, this enables a *write once - run wherever possible* development approach. The tool also facilitates energy savings on processors, such as JOP and Atmel's AVR, since TETASARTS allows the clock frequency of the target hardware to be set prior to the analysis. By adjusting this parameter, the lowest clock frequency, at which the system is still schedulable, can be found as demonstrated in [37].

Both JOP and HVM have been used in prototypical control systems such as the Minepump control system [27], the Real-Time Sorting Machine (RTSM) [11] and many student projects. The JOP has been used in industrial applications such as the Kippfahrleitung system for the Austrian Railways controlling up to 15 independent actuators [52].

In our opinion Safety Critical Java is now ready for more widespread use in robotics. It is supported by several academic and commercial implementations, and a number a tools can assist in the development of applications ensuring that they operate in time predictable manners.

The rest of this paper is organized as follows. In section I we describe the SCJ programming model. In section II we give an overview of three execution platforms for SCJ programs. In section III we describe a number of tools which can be used for analyzing timing properties of SCJ applications. Finally in section IV we draw conclusions and elaborate on future work.

I. THE SCJ REAL-TIME PROGRAMMING MODEL

Safety critical applications have different complexity levels. To cater for this the SCJ programming model is based on tasks grouped in missions, where a mission encapsulates a specific functionality or phase in the lifetime of the real-time system as a set of schedulable entities. The SCJ specification lets developers tailor the capabilities of the platform to the needs of the application through three compliance levels. The first level, 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 periodic and aperiodic event handlers, multiple missions, and a fixed-priority preemptive scheduler (FPS). Level 2 lifts restrictions on threads and supports nested missions. The development of SCJ applications at Level 0 is well described in [44]. In the remainder of this section we will focus on Level 1.

A. Missions

A mission encapsulates a specific functionality or phase in the lifetime of the real-time system as a set of schedulable entities. For instance, a flight-control system may be composed of take-off, cruising, and landing each of which can be assigned a dedicated mission. A schedulable entity handles a specific functionality and has release parameters describing the release pattern and temporal scope e.g. release time and deadline. The release pattern is either periodic or aperiodic.

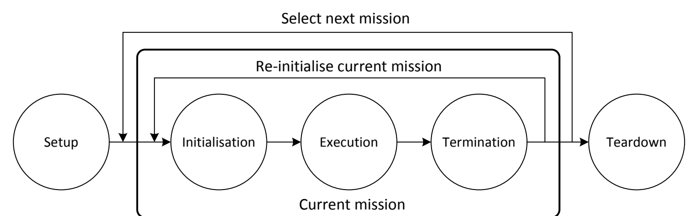


Fig. 1: Overview of the mission concept [38].

The mission concept is depicted in Figure 1 and contains five phases;

Setup where the mission objects are allocated. This is done during start-up of the system and is not considered time-critical.

Initialisation where all object allocations related to the mission or to the entire applications are performed.

Execution during which all application logic is executed and schedulable entities are set for execution according to a pre-emptive priority scheduler. This phase is time-critical.

Cleanup is entered if the mission terminates and is used for completing the execution of all schedulable entities as well as performing cleanup-related functionality. After this phase, the same mission may be restarted, a new is selected, or the Teardown phase is entered.

Teardown is the final phase in the lifetime of the application and comprises deallocation of objects and release of locks etc. This phase is not time-critical.

A *mission sequencer* is used for governing the order of the mission objects and can be customised to the application.

B. Memory Model

SCJ introduces a memory model based on the concept of *scoped memory* from the RTSJ, which circumvents the use of a garbage collected heap to ease verification of SCJ systems. The SCJ memory model is shown in Figure 2 and introduces three levels of memories;

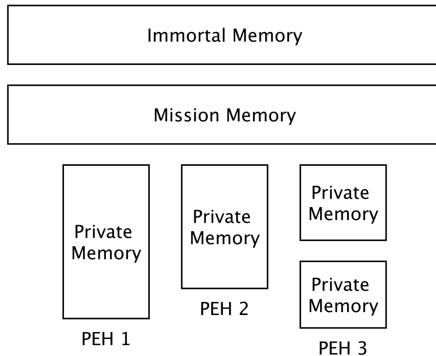


Fig. 2: The memory model in SCJ [38].

Private memory which is associated with each real-time event handler, which can be periodic (PEH) or aperiodic (APEH). The private memory exists for the entire duration of the handler. Upon task finish, the memory area is reset.

Mission memory is associated with every mission of the system and as such manages the memories of all real-time handlers part of the mission as well as objects that are shared among the handlers. When the mission completes execution, the mission memory is reset.

Immortal memory is the memory area that exists for the lifetime of the system.

Dynamic class loading is outside the scope of the SCJ specification. Hence, it is not necessary to reason about classes potentially being loaded over a network which would complicate timing analysis significantly. Furthermore, finalizers will not be executed and we make the assumption that Java Bytecode verification of class files has been done prior to the time-critical phase. The Predictable Java profile [9], being an alternative Java profile for hard real-time systems development, does allow the use of finalizers and [10] has demonstrated that timing analysis is possible. This may be important for systems written in a mix of Java and C++.

C. An SCJ Application

In this subsection we describe a hard real-time system implemented in SCJ based on a reduced version of the classical text-book control system example of a mine pump [15]. The purpose of the mine pump is to monitor a number of environmental properties in a mine to safely remove excess water using a water pump. It consists of two environmental properties being monitored: the water level in the mine and the methane level. When the water level rises to a predetermined level, the water pump is started, and when the water level drops to another predetermined level, the water pump is stopped. The water pump must not run if the methane levels exceed safe levels. These functionalities have temporal requirements stating the reaction times of the system required for safe operation such as timely stopping the water pump whenever a critical level of methane is reached.

Listing 1 shows the periodic event handler adhering to the SCJ profile.

```
1 PeriodicMethaneDetection
2   methaneDetection =
3     new PeriodicMethaneDetection(
4       new PriorityParameters(METHANE_DETECTION_PRIORITY
          ),
5       new PeriodicParameters(
6           new RelativeTime(0, 0),
7           new RelativeTime(PERIODIC_GAS_PERIOD, 0)),
8       new StorageParameters(
9           SCOPED_MEMORY_BACKING_STORE_SIZE,
10          NATIVE_STACK_SIZE,
11          JAVA_STACK_SIZE),
12      methaneSensor,
13      waterpumpActuator);
14
15 methaneDetection.register();
```

Listing 1: An SCJ handler for methane level [16].

An SCJ periodic event handler has a number of parameters: since the SCJ profile level 1 uses an FPS scheduler, evidently a priority must be specified. Furthermore, a release parameter specifies the start time, the relative initial time for the first release of the handler, and a further relative time gives the period. An instance of `StorageParameters` expresses memory-related constraints for the handler. The objects `methaneSensor` and `waterpumpActuator` are interfaces to a sensor and an actuator. The sensor observes the current methane level and the actuator starts and stops the water pump. When a handler instance has been created, it is set for being scheduled when the `register()` method is invoked.

```
1 public void handleEvent() {
2   waterpumpActuator.emergencyStop(
3     methaneSensor.isCriticalMethaneLevelReached()
4   );
5 }
```

Listing 2: Detecting the methane level [16].

Listing 2 shows the event handling method of the periodic event handler `PeriodicMethaneDetection`.

The actual prototype consists of two parts: the physical plant and the control software. Lego is used to construct the physical plant together with Lego NXT sensors and actuators connected to a JOP board or a board with an AVR ATmega2560. The control software comprises two periodic and two sporadic real-time tasks written in Java. The periodic tasks are responsible for monitoring the methane and water levels. The sporadic tasks are released whenever either the low or the high level has been reached. More details can be found in [16].

II. Real-Time Execution Platforms

The SCJ programming model provides a structuring framework for applications with hard-real-time requirements. Next such applications need an execution platform. For applications written in C this is usually a hardware processor. However, Java applications are typically translated into Java Bytecodes which are then either interpreted or further translated into native code before execution, also called ahead-of-time (AOT) execution or during, also called just-in-time (JIT) execution. This approach entails a time predictable implementation of each Java Bytecode. In this section we will describe the Java Optimized Processor (JOP) [51], the FijiVM ahead-of-time compiler for Java Bytecode programs [43] and The Hardware near Virtual Machine (HVM) [59], [32]. There are commercial implementations of the JVM supporting the SCJ programming model. These include the FijiVM, JamaicaVM [1] and PicoPERC [42].

A. Java Optimized Processor

The simplest way to ensure a time predictable execution of each Jave Bytecode is to implement the JVM in hardware. This is the approach taken by the JOP [51]. The JOP is implemented on an FPGA (Altera Cyclone EP1C6Q240 or EP1C12Q240). The JOP has its own micro code instruction set with most Java Bytecodes having a one-to-one mapping. However, some Java Bytecodes are more complex and are implemented as a sequences of JOP micro codes, some are even implemented in Java. However, the end result is that for each Java Bytecode its execution can be bound and its WCET be determined.

Important for WCET analysis of programs executing on the JOP is that the JOP does not feature data caches, but it features a method cash and its use must be taken into account for tight bounds of WCET.

The JOP is usually hosted on a board which comes in two configurations; The Baseio, which provides a complete Java Processor system with Internet connection for JOP with network connection via a CS8900 Ethernet controller with RJ 45 connector. Java sources for CS8900 driver and a simple TCP/IP stack are available for JOP. The Simpexp, which is a cheap IO extension to get started with JOP, contains a linear 3.3V regulator and serial connector. The JOP can interface to sensors and motors of the LEGO Mindstorms series and thus the JOP can substitute the LEGO RCX.

B. FijiVM

FijiVM is an ahead-of-time compiler for Java Bytecode programs [43]. FijiVM has sufficiently low memory demands and is applicable for embedded systems. The FijiVM parses Bytecodes in the Java 1.6 or earlier format and generates ANSI C code. The generated C code is then automatically passed to a C compiler, typically GCC, for the target platform. The FijiVM compiler does extensive high-level optimizations prior to generating C, as the C compiler cannot perform some high-level optimizations as effectively on C code generated from Java as a Java-optimized compiler could. Such optimizations include control flow optimizations, such as Intra-procedural and whole-program type propagation; Devirtualization, turning virtual calls into direct calls and Virtualization, turning interface calls into virtual calls; exception optimizations; Null pointer check elimination; Array bounds check removal; locking and memory optimizations; Inlining; Copy propagation; Constant folding and Tail duplication.

The FijiVM generates a stand-alone executable which, however, is dependent on libraries that are standard on POSIX-like OSs, such as Linux (libc, libpthread, and libm). The FijiVM has support for embedded processors such as ARM and ERC32 or more powerful processors such as PowerPC and x86/x86_64.

C. The Hardware near Virtual Machine

The HVM [59], [32] is a lean JVM implementation intended for use in resource-constrained embedded devices with as low as 256 KB ROM and 20 KB RAM. It features both iterative interpretation, Java-to-C compilation (AOT), and a hybrid of the two.

The HVM employs *JVM specialisation*; a JVM is produced specifically for hosting the Java Bytecode program of a given application. This is done using the ICECAP-TOOLS Eclipse-plugin, which analyzes the Java Bytecode program and produces an executable for the target platform. The analyzes and transformations can be extended, and it incorporates a number of static analyzes for improving performance of the JVM and for reducing its size. This includes receiver-type analysis for potentially devirtualising method calls and intelligent class linking which computes a conservative set of classes and methods that are used in the application. Only this set will be embedded in the final HVM executable. It also conservatively estimates the set of Java Bytecodes that will actually be used. Those that are not, are omitted from the final executable.

The HVM is self-contained and does not rely on the presence of an OS or a C standard library. The overall structure of an SCJ application running on top of the HVM using the accompanying SCJ implementation (HVM-SCJ [59]) as well as the ICECAP-TOOLS SDK is shown in Figure 3. Porting the HVM to new target platforms (including SCJ support) is a matter of implementing the *HW Interface* and the *VM Interface*.

The HVM implements SCJ level 0, 1 and 2 [62]. It has support for multicore[1] and Hardware Objects [33], an object-oriented abstraction of low-level hardware devices such as interrupts and I/O registers which can be handled from Java space. A related feature is *native variables*, which allow for access to certain variables in the JVM from Java space.

The HVM has been ported to Atmel AVR ATmega2560 microcontroller, Arduino and Lego EV3.

III. Timing Analysis Tools

Rigorous verification is essential for safety critical embedded hard real-time systems needing to comply with tight timing constraints. Of special interest, is the verification of the system being *schedulable*, i.e. verifying that all real-time tasks under the conditions of the employed scheduling policy, finish before their respective deadlines in all circumstances. The Worst and Best Case Execution Time (WCET and BCET) often play an integral role in this relation – especially the former,

[1] https://github.com/zs673/Multiprocessor-icecap-SCJ-RTE

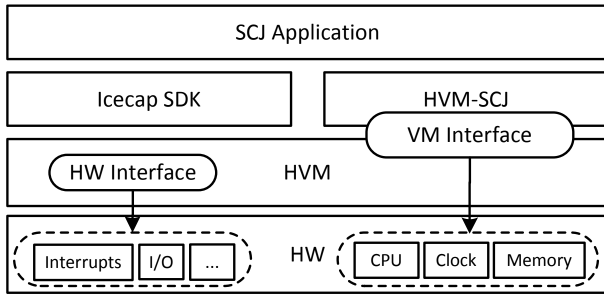


Fig. 3: Constituents of an SCJ application on HVM [38].

which has applications in traditional methods for verification of schedulability such as response time analysis [14].

For systems written in C or rather a suitable subset of C, there are many such analysis tools, both academic and commercial [4], [34], [20], [45], [17], [23], [29], [47], [26]. These tools vary in the platforms they support, in the way they analyze programs and which restrictions they place on the analyzed programs, but as stated in [61] "To avoid having to solve the halting problem, all programs under analysis must be known to terminate. Loops need bounded iteration counts and recursion needs bounded depth". The amount of required annotations is reduced by analysis, such as automatic loop-bound and array-call recognition.

Analyzing timing properties for Java programs is challenging primarily due to the fact that Java is usually translated to Java Bytecode, which is then interpreted by a JVM or further translated into native code, sometimes via a compilation to C. This level of indirection complicates formal analysis as both program and JVM have to be taken into account for a given hardware platform; some of this complexity can, however, be reduced by a hardware implementation of the JVM such as JOP.

Recently a number of analysis tools have been developed including WCA [54], SARTS [11], TetaJ [27], TETASARTS [41] and SymRT [40]. In this section we give an overview of three tools; WCA, TETASARTS and SymRT.

A. WCA

WCET Analyzer (WCA) [55], [30] is a static code analysis tool for conducting WCET analysis of Java Bytecode executed on the JOP. As described earlier the JOP is a hardware implementation of the JVM which facilitates known execution times of each Java Bytecode. The relative simplicity and predictability of the JOP architecture and, in particular, the use of a method cache instead of more general cache disciplines, makes it relatively easy to perform precise WCET analysis. WCA employs two distinct strategies for WCET analysis; one is the Implicit Path Enumeration Technique (IPET) [35] and the other models the real-time application using timed automata in the verification tool UPPAAL [5]. The rationale behind supporting two different strategies is that the two represent a trade off between estimation time and precision. In WCA, the IPET strategy yields WCET estimates relatively fast, while the model-based strategy results in more precise estimates at the cost of a relatively long verification time. The precise WCET estimate is a consequence of the model representing the detailed behaviour of the system, especially the cache model. Common to both WCET estimation strategies is the Control-Flow Graph (CFG) of the application which is constructed by consulting the Java class files using the Byte Code Engineering Library. For the IPET strategy, WCA transforms the CFG into an integer linear programming problem which is solved using the linear programming solver lp_solve [7] resulting in a WCET estimate. In the modelbased strategy, the CFG is directly transformed into timed automata models for UPPAAL. Currently, WCET estimates using the model-based strategy are computed by making an initial guess of WCET, which can be based on the estimate derived using IPET. Afterwards, UPPAAL verifies whether the timed automata are verifiable within the guessed time and, afterwards, the estimate is gradually refined using a binary search tactic. For unbounded loops, WCA introduces comment-based annotations of source code which make explicit the iteration count of the particular loop. Alternatively, WCA provides the option of using data-flow analysis for extracting these. Obviously not all bounds can be extracted as part of static code analysis and in such cases the programmer needs to insert annotations. Furthermore, WCA performs receiver type analysis to increase the precision of the WCETs in case of dynamic method dispatch. Besides printing the resulting WCET estimate to standard output, WCA conveniently generates a detailed HTML report containing a visual representation of the CFG and timings of individual methods including their cache misses.

B. TETASARTS

TETASARTS started as an amalgamation of the SARTS [11] and TetaJ [27] tools and is today an open-source collection of timing analysis tools which operate on a timing model amenable to model checking using UPPAAL. Figure 4 shows the major components of the toolchain and their interactions.

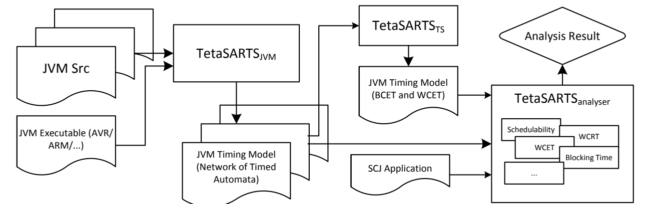


Fig. 4: Overview of the TETASARTS toolchain [38].

Reasoning about the timeliness of the system or any other real-time aspect, requires both the real-time application as well as the underlying execution environment to be modeled. Generating an appropriate model of the latter is the purpose of the TETASARTS$_{\text{JVM}}$[2] tool.

TETASARTS$_{\text{JVM}}$ generates a timing model as a Network of Timed Automata (NTA), the modeling formalism of the UPPAAL model checker. A Timed Automaton (TA) is a finite state machine extended with real-valued clocks. An NTA is the parallel composition of a number of TAs sharing clocks and actions (for details of the semantics, see [6]).

[2]TETASARTS$_{\text{JVM}}$, HVM$_{\text{TP}}$, and generated models are available on the project website: http://people.cs.aau.dk/~luckow/hvmtp/

TETASARTS$_{\text{JVM}}$ builds a TA that captures the control-flow for each of the supported Java Bytecodes. The tool processes the JVM executable such that compiler optimisations, transformations etc. are accounted for when reconstructing the control-flow. TETASARTS$_{\text{JVM}}$ conducts loop identification analysis and expects loop bounds to either be provided interactively at construction time or as comment-style annotations in the source code of the JVM. The tool allows specifying code regions constituting Java Bytecode implementations. The regions are created by embracing the Java Bytecode implementations with macros: BEGIN_JBC(X) and END_JBC(X) mark the beginning and end, respectively, of Java Bytecode X. The macros generate instrumentation code in the binary, which is used by TETASARTS$_{\text{JVM}}$ for reconstructing the control-flow. Listing 3 shows the implementation of the i2l Java Bytecode and the region specification.

```
1  case I2L_OPCODE: {
2  #if defined(INSTRUMENT)
3    BEGIN_JBC(I2L_OP);
4  #endif
5    int32 lsb = *(--sp);
6    if(lsb < 0) {
7      *sp++ = -1;
8    } else {
9      *sp++ = 0x0;
10   }
11   *sp++ = lsb;
12   method_code++;
13 #if defined(INSTRUMENT)
14   END_JBC(I2L_OP);
15 #endif
16 }
```

Listing 3: i2l [38].

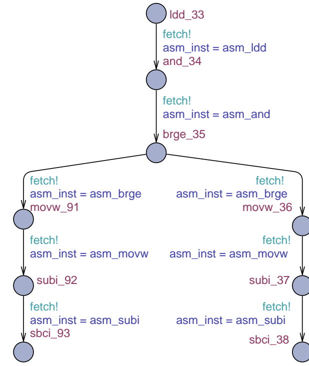
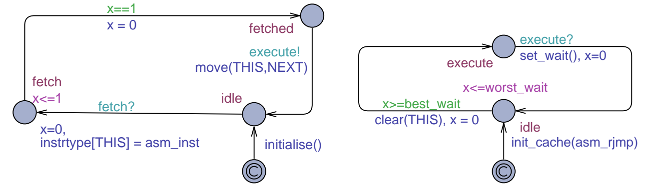


Fig. 5: TA excerpt of i2l [38].

Figure 5 shows an excerpt of the corresponding TA generated by TETASARTS$_{\text{JVM}}$. As an example, note how the location labelled *brge_35* branches to *movw_91* and *movw_36*. This captures the if-statement in line 6 of Listing 3. Each transition simulates the timing behavior of executing the instruction assigned to the UPPAAL variable *asm_inst*. In general, this explicit modeling of system behavior does not scale to large systems due to the inherent problem of state space explosion. However, it is important to note that in our case, the behavior of each Java Bytecode in isolation is sufficiently small to allow model checking to be feasible. The Java Bytecode implementations we are analyzing, range from just a few lines of C code in the simplest case, to a few hundred lines of C code producing ∼200 and ∼9000 machine instructions, respectively.

We denote the composition of the generated TAs, which yield the NTA, as the *JVM NTA*. The timing behavior depends on the hardware used; in this example the AVR ATmega2560 microcontroller whose behavior is captured by the TAs shown in Figure 6 collectively referred to as the *HW NTA*. In addition TETASARTS$_{\text{JVM}}$ supports the ARM7 and ARM9 models from METAMOC [23] and can be extended to support different hardware by providing TAs modeling the hardware. The *fetch* channel is used for hand-shake synchronisation between the JVM NTA and the pipeline fetch stage in Figure 6a; *asm_inst* communicates the instruction to be simulated in the HW NTA. Similarly, the *execute* channel establishes the communication between the fetch stage TA and the execute stage TA in Figure 6b. In both TAs, x is a clock variable simulating the



(a) Pipeline fetch stage. (b) Pipeline execute stage.

Fig. 6: Hardware TA models from METAMOC [23].

instruction processing time in the respective pipeline stage. In the execute stage TA, *worst_wait* and *best_wait* are set according to the worst and best case clock cycle execution times of the simulated instruction. Note the call *init_cache(asm_rjmp)*; it initializes the pipeline with the temporal behavior of the *rjmp* instruction. This is necessary to ensure a safe timing model, since the pipeline will be filled with this instruction prior to executing the first instruction of any of the Java Bytecodes.

Synthesizing the JVM NTA and the HW NTA yields a complete *JVM Timing Model* (see Figure 4) that simulates the timing behavior of the JVM. The JVM Timing Model can subsequently be used directly by further synthesising it with a model of an SCJ application on which various timing related analyses can be applied such as schedulability analysis and WCRT analysis using the TETASARTS$_{\text{ANALYSER}}$ tool (see [37], [41], [39] for more details).

The JVM Timing Model can also be used for representing the timing behavior of the JVM more abstractly using the TETASARTS$_{\text{TS}}$ tool (see Figure 4). For all Java Bytecode TAs, the tool determines the WCET and BCET using the sup- and inf-query extensions of the UPPAAL model checker. Sup- and inf-queries perform a state space exploration, and outputs the maximum and minimum values, respectively, of the specified clock-variables.

TETASARTS takes the interactions between tasks into account during schedulability analysis, thus systems deemed unschedulable using traditional response time analysis may be deemed schedulable using model checking as demonstrated in [11]. TETASARTS also allows the clock frequency of the target hardware to be set prior to the analysis, thus making it possible to determine the lowest clock frequency at which the system is still schedulable as demonstrated in [37].

C. SymRT

SymRT [40] is a tool based on a combination of symbolic execution [31], [19] and real-time model checking that generates a precise control-flow model from the symbolic execution trees obtained with a symbolic execution of the program. Each tree characterizes the set of *feasible* execution paths (up to some bound) of the analyzed task and yields a precise timing model.

Symbolic execution is used for generating a safe and tight timing model of the analyzed system capturing the feasible execution paths. This timing model is combined with execution environment models capturing the timing behavior of the target host platform including the JVM and complex hardware

features such as caching. The complete timing model is a NTA and directly facilitates safe estimates of Worst (and Best) Case Execution Time to be determined using the UPPAAL model checker. Furthermore, the integration of these techniques into the TETASARTS tool facilitates reasoning about additional timing properties such as the schedulability of periodically and sporadically released Java real-time tasks (under specific scheduling policies), Worst Case Response Time, and more.

The program model is built *modularly*. The timing behavior of the execution environment is obtained from *environment models* capturing the timing behavior of the JVM and hardware of the target platform. This technique generates the complete timing model, based on a configuration, as an NTA amenable for model checking using the UPPAAL [5] model checker. The NTA model can be used for estimating WCET and BCET. Furthermore, it generalizes to verification of properties expressible in Timed Computation Tree Logic (TCTL). In contrast to previous tools [27], [41], which provide limited feedback, SymRT can also generate *witness traces* that expose the reported behaviour, useful for debugging and program understanding.

Both symbolic execution and model checking have issues with scalability due to the large number of paths respectively states to explore. SymRT addresses this by using a "per task" symbolic execution, leveraging the SCJ programming model, that groups code into missions consisting of relatively short tasks. Furthermore, when using timing models of the target execution environment, the generated TA of the program is at basic block level, which significantly reduces the state space size.

Clearly there are systems for which analysis is intractable. This is for instance the case when attempting to do a full schedulability analysis for the Real-Time Sorting Machine (RTSM) [11] on HVM on the AVR processor. This application consists of 1 kloc. However, it is not the number of lines of code that limits the analysis, but its complexity, especially the number of branch points in each task and the number of tasks in the application, as the size of the UPPAAL model grows exponentially with the number of components in the NTA. However, since schedulability is viewed as a reachability problem, it may be possible to translate it into the subset of the UPPAAL modeling language supported by the opaal+LTSmin system [24]. In [22] opaal+LTSmin demonstrates a speedup of 40 on a 48 core machine compared to UPPAAL. Future work will investigate this direction.

D. Comparison of tools

This section summarizes a comparison of the WCET (and BCET) estimates obtained from WCA, TETASARTS and SYMRT, reported in more details in [40].The comparison uses as examples the Java implementations (obtained from the JOP distribution[3]) of a subset of the algorithms from the Mälardalen WCET benchmark suite [28]: Bubble Sort, Quick Sort, Insertion Sort and Binary Search. For the sorting algorithms, the array is initialized with symbolic values. For Binary Search, the search key is symbolic. Note that there was no need to provide loop bound annotations in any of the examples nor did they reach the default search depth (corresponding to 100 branches) during analysis.

For the analysis two configurations of the execution environment was used; (1) JOP and (2) HVM[4] [59] running on the AVR ATmega2560. The same configuration of the execution environment is used across the tools e.g. WCA and SYMRT have been configured with read and write wait cycles set to 1 and 2 (and cache configuration is the same).The analysis also compared measured BCET and WCET obtained by using inputs yielding the best and worst case behavior (e.g. for Bubble Sort a sorted and unsorted list were used). Furthermore, the JOP simulator was used to to read the cycle count before the first instruction is executed of the target method and after the return instruction. The measurements for HVM+AVR have been obtained in a similar way by using the debugging facilities of Atmel Studio 6. For this set of experiments, a laptop with an Intel Core i7-2620M CPU @ 2.70GHz with 8 GB of RAM was used. The peak memory consumption for symbolic execution is 500-700 MB for all examples. UPPAAL peaks at 50-200 MB during model checking. Table I shows the results for the comparison on JOP.

First note that all estimates are *safe* i.e. ($\text{BCET}_{symrt} \leq \text{BCET}_{\text{m}}$ and $\text{WCET}_{symrt} \geq \text{WCET}_{\text{m}}$) and that the precision of SYMRT is better (and in one case equally as good) as the other tools. The major contributor to the pessimistic results of TETASARTS and WCA are that they over-approximate the iterations of nested loops with interdependencies. The analysis time using SYMRT is however longer, which is due to symbolic execution. Also note that for e.g. Quick Sort, it is relatively difficult to exercise and measure the path yielding the worst case behavior since it depends on the pivot element selection.

Table II shows the comparison when using the HVM and an AVR ATmega2560.

Again all estimates produced by SYMRT are safe and more precise than TETASARTS. For this first set of experiments (including the results obtained for JOP), the analysis time is largely attributed symbolic execution. In all cases, model checking using UPPAAL takes less than a second.

We compare the schedulability analysis of SYMRT with TETASARTS using the Minepump control system [14], [27], the Real-Time Sorting Machine (RTSM) [11] and a variant of MD5SCJ [41]. For this set of experiments we used an application server with an Intel Xeon X5670 @ 2.93GHz CPU and 32 GB of RAM. The results are shown in Table III. We also conducted the analysis on a version of the Lift real-time system from Jembench [53] with 18 tasks. TETASARTS has not been able to construct the models for this. The *TD* subscript denotes that a Timing Scheme with fixed execution times for all the Java Bytecodes has been used instead of modeling their behavior as an NTA.

In all cases, the systems have been deemed schedulable, and the results show that the analysis times and memory consumptions are lower when using SYMRT. We also tried e.g. RTSM with HVM+AVR, but the complexity of the resulting models regardless of the tool used, is too big, which can

[3]Available for download at http://www.jopdesign.com/

[4]Available for download at http://icelab.dk/

System	SymRT (JPF-Symbc-rt) BCET [cycles]	SymRT WCET [cycles]	SymRT An. Time [seconds]	TetaSARTS WCET [cycles]	TetaSARTS An. Time [seconds]	WCA WCET [cycles]	WCA An. Time [seconds]	Measured $BCET_m$ [cycles]	Measured $WCET_m$ [cycles]
Binary Search	136	818	1	927	1	818	1	138	722
Bubble Sort	653	1,253	51	1,770	2	1,553	1	653	1,253
Quick Sort	1,425	2,638	510	18,749	5,375	20,275	1	1,425	1,895
Insertion Sort	774	2,586	21	4,600	1	4,296	1	774	2,586

TABLE I: Comparison of SymRT, TetaSARTS, and WCA [40].

System	SymRT (JPF-Symbc-rt) BCET [cycles]	SymRT WCET [cycles]	SymRT An. Time [seconds]	TetaSARTS WCET [cycles]	TetaSARTS An. Time [seconds]	Measured BCET [cycles]	Measured WCET [cycles]
Binary Search	3,991	65,046	2	70,153	2	4,140	23,262
Bubble Sort	19,514	93,380	50	287,526	31	19,754	37,388
Quick Sort	42,651	151,784	589	133,134	228	43,251	50,437
Insertion Sort	20,351	182,099	21	244,680	4	22,625	70,028

TABLE II: Comparison of SymRT and TetaSARTS for systems running on the HVM and AVR [40].

System	Exec. Env.	Analysis time SymRT	Analysis time TetaSARTS	Memory SymRT	Memory TetaSARTS
Minepump	HVM+AVR	14h 12m	15h 25m	16274 MB	17933 MB
Minepump	HVM+AVR_{TD}	< 1s	2s	8 MB	11 MB
Minepump	JOP	< 1s	1s	5 MB	11 MB
RTSM	HVM+AVR_{TD}	< 1s	1m 2s	7 MB	17 MB
RTSM	JOP	< 1s	5s	6 MB	15 MB
MD5SCJ	HVM+AVR_{TD}	< 1s	8s	7 MB	17 MB
MD5SCJ	JOP	< 1s	1m 23s	5 MB	47 MB
Lift	HVM+AVR_{TD}	33m 6s	–	5897 MB	–
Lift	JOP	15m 43s	–	6037 MB	–

TABLE III: Comparison of TetaSARTS and SymRT [40].

be attributed the JVM NTA, which largely dominates the complexity. In this case, Uppaal runs out of memory.

IV. Conclusion

In this paper we have presented the Safety Critical Java programming model, some of its implementations in the form of the JOP, FijiVM and HVM and some of the tools, especially WCA, TetaSARTS tool suite and SymRT, allowing programmers to analyze their SCJ applications for correct time behaviour. Furthermore, we have argued the suitability of SCJ for use in robotics applications with hard-real-time constraints.

Small, but realistic SCJ applications, such as the Minepump control system, the Real-Time Sorting Machine (RTSM) and a variant of MD5SCJ, as well as a number of applications from the Mälardalen WCET Benchmarks, have been implemented and analyzed using the above mentioned tools. A number of student projects have used either the JOP or the HVM on either Arduino or Lego EV3 to learn about robotics programming using robots build in Lego Mindstorm. The JOP has been used in industrial applications such as the Kippfahrleitung system for the Austrian Railways controlling up to 15 independent motors [52]. Although not written SCJ, larger systems with real-time constraints have been developed in Java. [2] presents the first use of Real-time Java in avionics in the context of control software for a ScanEagle Unmanned Aerial Vehicle, and [56] presents the Use of PERC Pico in the AIDA Avionics Platform. Our own work include the performance analysis of different components of a NASA tactical layer solution for planes, T-TSAFE, currently focusing on the conflict detection and conflict resolution algorithms.

Clearly SCJ is not the solution for all robotics applications. The programming model of tasks and missions, and especially the scoped memory model, is rather restrictive. All the mentioned JVM implementations have real-time garbage collection implementations available, and it would thus be possible to dispense with the scoped memory model. To the best of our knowledge, at present, none of these garbage collectors have been analyzed for time predictability and thus cannot be used in systems needing to comply with standards such as DO-178C, ISO-26262, IEC-61508 and EN-50128. However, we expect this to be just a matter of time and hard work.

Many robotics systems will have components that are not time critical. Currently SCJ does not cater well for such mixed criticality systems. RTSJ caters for such mixed criticality systems, however, at the expense of analyzability. We envision that recent developments in compositional schedulability analysis [13] could be integrated with the SCJ programming model, basically allowing different missions to have different scheduling policies and using the SCJ Level 2 notion of nested missions to implement hierarchies of components with different criticality levels. This would allow components with no real-time requirements to execute in their own missions as long as there is time budget for the time critical components to execute their tasks. The HVM facilitates a tight integration with (legacy) code in C, i.e. handlers in Java can directly be called from handlers in C and visa versa, clearly at the expense of more complex analysis, however, some some systems it is not possible to port all parts of the code to Java. We envision that the techniques of I/O automatas [25] used in the ECDAR tool, analysis of C code using METAMOC [23] and the notion of schedulability abstraction [8] could be combined to provide

a framework for analysis of such mixed applications.

Many advanced robotic systems are programmed using rule-based systems. Another, direction for future work, would be to cater for such robotics systems by implementing a time predictable rule match algorithm in SCJ, e.g. implementing a variant of the rete-algorithm [18] used in some advanced robotics applications.

References

[1] Aicas. *JamaicaVM User Manual: Java Technology for Critical Embedded Systems*, 2010.

[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. *ACM Trans. Embed. Comput. Syst.*, 7(1):5:1–5:49, December 2007.

[3] Atego. Atego Home, 2013. http://atego.com/.

[4] Clément Ballabriga, Hugues Cassé, Christine Rochange, and Pascal Sainrat. OTAWA: An Open Toolbox for Adaptive WCET Analysis. In *Software Technologies for Embedded and Ubiquitous Systems*, pages 35–46. Springer, 2011.

[5] Johan Bengtsson, Kim Larsen, Fredrik Larsson, Paul Pettersson, and Wang Yi. UPPAAL – a Tool Suite for Automatic Verification of Real-Time Systems. In *Hybrid Systems*. 1996.

[6] Johan Bengtsson and Wang Yi. Timed automata: Semantics, algorithms and tools. In *Lectures on Concurrency and Petri Nets*, Lecture Notes in Computer Science. 2004.

[7] M Berkelaar. lp_solve reference guide.[online].", 2014.

[8] T. Bogholm, B. Thomsen, K.G. Larsen, and A. Mycroft. Schedulability analysis abstractions for Safety Critical Java. In *Object/Component/Service-Oriented Real-Time Distributed Computing (ISORC), 2012 IEEE 15th International Symposium on*, pages 71 –78, april 2012.

[9] Thomas Bøgholm, René R. Hansen, Anders P. Ravn, Bent Thomsen, and Hans Søndergaard. A predictable Java profile: Rationale and implementations. In *Proceedings of the 7th International Workshop on Java Technologies for Real-Time and Embedded Systems*, JTRES '09, pages 150–159, New York, NY, USA, 2009. ACM.

[10] Thomas Bøgholm, René R. Hansen, Anders P. Ravn, Bent Thomsen, and Hans Søndergaard. Schedulability analysis for Java finalizers. In *Proceedings of the 8th International Workshop on Java Technologies for Real-Time and Embedded Systems*, JTRES '10, pages 1–7, New York, NY, USA, 2010. ACM.

[11] Thomas Bøgholm, Henrik Kragh-Hansen, Petur Olsen, Bent Thomsen, and Kim 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 '08, pages 106–114, New York, NY, USA, 2008. ACM.

[12] Gregory Bollella and James Gosling. The real-time specification for java. *IEEE Comp.*, 2000.

[13] Jalil Boudjadar, Kim Guldstrand Larsen, Jin Hyun Kim, and Ulrik Nyman. *Compositional Schedulability Analysis of An Avionics System Using UPPAAL*. 2014.

[14] Alan Burns and Andy Wellings. *Real-Time Systems and Programming Languages: Ada 95, Real-Time Java, and Real-Time POSIX*. 2009.

[15] Alan Burns and Andy Wellings. *Real-Time Systems and Programming Languages: ADA 95, Real-Time Java, and Real-Time POSIX*. Addison-Wesley Educational Publishers Inc., Boston, MA, USA, 4th edition, 2009.

[16] Thomas Bøgholm, Christian Frost, RenéRydhof Hansen, CasperSvenning Jensen, KasperSøe Luckow, AndersP. Ravn, Hans Søndergaard, and Bent Thomsen. Towards harnessing theories through tool support for hard real-time Java programming. *Innovations in Systems and Software Engineering*, 9(1):17–28, 2013.

[17] Mälardalen University Real-Time Research Center. SWEET (SWEdish Execution Time tool). http://www.mrtc.mdh.se/projects/wcet/sweet/.

[18] Yanik Kim Challand. A real time expert system: Adapting match algorithms and implementing a tailored rule language. Master's thesis, Aalborg University, June 2011.

[19] Lori A. Clarke. A System to Generate Test Data and Symbolically Execute Programs. *IEEE Trans. Software Eng.*, 1976.

[20] Antoine Colin and Isabelle Puaut. A modular and retargetable framework for tree-based WCET analysis. In *Real-Time Systems, 13th Euromicro Conference on, 2001.*, pages 37–44. IEEE, 2001.

[21] Java Robotics Community. Java robotics community. https://community.java.net/community/robotics.

[22] Andreas E Dalsgaard, Alfons Laarman, Kim G Larsen, Mads Chr Olesen, and Jaco Van De Pol. Multi-core reachability for timed automata. In *Formal Modeling and Analysis of Timed Systems*, pages 91–106. Springer, 2012.

[23] Andreas E. Dalsgaard, Mads Chr. Olesen, Martin Toft, René Rydhof Hansen, and Kim Guldstrand Larsen. METAMOC: Modular Execution Time Analysis using Model Checking. In *10th International Workshop on Worst-Case Execution Time Analysis*, 2010.

[24] AndreasEngelbredt Dalsgaard, RenéRydhof Hansen, KennethYrke Jørgensen, KimGulstrand Larsen, MadsChr. Olesen, Petur Olsen, and Jiří Srba. opaal: A Lattice Model Checker. In Mihaela Bobaru, Klaus Havelund, GerardJ. Holzmann, and Rajeev Joshi, editors, *NASA Formal Methods*, volume 6617 of *Lecture Notes in Computer Science*, pages 487–493. Springer Berlin Heidelberg.

[25] Alexandre David, Kim G. Larsen, Axel Legay, Ulrik Nyman, and Andrzej Wasowski. Timed i/o automata: A complete specification theory for real-time systems. In *Proceedings of the 13th ACM International Conference on Hybrid Systems: Computation and Control*, HSCC '10, pages 91–100, New York, NY, USA, 2010. ACM.

[26] Christian Ferdinand, Reinhold Heckmann, and Bärbel Franzen. Static memory and timing analysis of embedded systems code. In Perry Groot, editor, *Proceedings of VVSS2007 - 3rd European Symposium on Verification and Validation of Software Systems, 23rd of March 2007, Eindhoven*, number TUE Computer Science Reports 07-04, 2007.

[27] Christian Frost, Casper Svenning Jensen, Kasper Søe Luckow, and Bent Thomsen. WCET analysis of Java bytecode featuring common execution environments. In *9th International Workshop on Java Technologies for Real-Time and Embedded Systems*, 2011.

[28] Jan Gustafsson, Adam Betts, Andreas Ermedahl, and Björn Lisper. The Mälardalen WCET Benchmarks - Past, Present and Future. In *Proceedings of the 10th International Workshop on Worst-Case Execution Time Analysis*, July 2010.

[29] Niklas Holsti and Sami Saarinen. Status of the Bound-T WCET tool. *Space Systems Finland Ltd*, 2002.

[30] Benedikt Huber and Martin Schoeberl. Comparison of implicit path enumeration and model checking based wcet analysis. In *OASIcs-OpenAccess Series in Informatics*, volume 10. Schloss Dagstuhl-Leibniz-Zentrum für Informatik, 2009.

[31] James C. King. Symbolic Execution and Program Testing. *Commun. ACM*, 1976.

[32] Stephan Korsholm. Hvm, 2013. http://www.icelab.dk.

[33] Stephan Korsholm, Anders P. Ravn, Christian Thalinger, and Martin Schoeberl. Hardware objects for Java. In *Proceedings of the 11th IEEE International Symposium on Object/component/serviceoriented Real-time distributed Computing (ISORC 2008*, 2008.

[34] Xianfeng Li, Yun Liang, Tulika Mitra, and Abhik Roychoudhury. Chronos: A Timing Analyzer for Embedded Software. *Science of Computer Programming*, 69(1):56–67, 2007.

[35] Yau-Tsun Steven Li and Sharad Malik. Performance Analysis of Embedded Software Using Implicit Path Enumeration. In *Proceedings of the 32Nd Annual ACM/IEEE Design Automation Conference*, DAC '95, pages 456–461, New York, NY, USA, 1995. ACM.

[36] 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.

[37] K. S. Luckow, T. Bøgholm, and B. Thomsen. Supporting Development of Energy-Optimised Java Real-Time Systems using TetaSARTS. In *WiP Proceedings of the 19th Real-Time and Embedded Technology and Application Symposium*, 2013.

[38] K. S. Luckow, B. Thomsen, and S. E. Korsholm. HVM-TP: A Time Predictable and Portable Java Virtual Machine for Hard Real-

Time Embedded Systems. In *12th International Workshop on Java Technologies for Real-Time and Embedded Systems*, 2014. To appear.

[39] K. Søe Luckow, Thomas Bøgholm, Bent Thomsen, and Kim Guldstrand Larsen. TetaSARTS: Modular Timing and Performance Analysis of Safety Critical Java Systems. *Concurrency and Computation: Practice and Experience*, 2014. In Submission.

[40] Kasper Søe Luckow. *Platforms and Model-Based Analyses for Real-Time Java*. PhD thesis, Department of Computer Science, Aalborg University, 2014, http://people.cs.aau.dk/~luckow/thesis.pdf.

[41] Kasper Søe Luckow, Thomas Bøgholm, Bent Thomsen, and Kim Guldstrand 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, pages 11–20, New York, NY, USA, 2013. ACM.

[42] Kelvin Nilsen. Differentiating features of the perc virtual machine, 2009.

[43] 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 '09, pages 110–119, New York, NY, USA, 2009. ACM.

[44] 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 '10, pages 95–101, New York, NY, USA, 2010. ACM.

[45] Adrian Prantl, Markus Schordan, and Jens Knoop. TuBound - A Conceptually New Tool for Worst-Case Execution Time Analysis. In Raimund Kirner, editor, *8th International Workshop on Worst-Case Execution Time Analysis (WCET'08)*, volume 8 of *OpenAccess Series in Informatics (OASIcs)*, Dagstuhl, Germany, 2008. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik. also published in print by Austrian Computer Society (OCG) with ISBN 978-3-85403-237-3.

[46] Jerry Pratt and Ben Krupp. Design of a bipedal walking robot. In *SPIE Defense and Security Symposium*, pages 69621F–69621F. International Society for Optics and Photonics, 2008.

[47] RapiTime. Rapitime wcet tool homepage. http://www.rapitasystems.com.

[48] LCC RidgeSoft. Robojde java-enabled robotics software development environment, 2014. http://www.ridgesoft.com/robojde/robojde.htm.

[49] Sven Gestegård Robertz, Roger Henriksson, Klas Nilsson, Anders Blomdell, and Ivan Tarasov. Using real-time java for industrial robot control. In *Proceedings of the 5th international workshop on Java technologies for real-time and embedded systems*, pages 104–110. ACM, 2007.

[50] Andreas Schierl, Andreas Angerer, Alwin Hoffmann, Michael Vistein, Wolfgang Reif, and I Vision. Using Java for real-time critical industrial robot programming. 2012.

[51] Martin Schoeberl. JOP: A Java Optimized Processor. In *Proceedings of Java Technologies for Real-Time and Embedded Systems*, 2003.

[52] Martin Schoeberl. Using a java optimized processor in a real world application. In *WISES*, pages 165–176, 2003.

[53] Martin Schoeberl, Thomas B. Preusser, and Sascha Uhrig. The Embedded Java Benchmark Suite JemBench. In *Proceedings of the 8th International Workshop on Java Technologies for Real-Time and Embedded Systems*, JTRES '10, pages 120–127, New York, NY, USA, 2010. ACM.

[54] Martin Schoeberl, Wolfgang Puffitsch, Rasmus Ulslev Pedersen, and Benedikt Huber. Worst-Case Execution Time Analysis for a Java Processor. *Softw.: Prac. and Exp.*, 2010.

[55] Martin Schoeberl, Wolfgang Puffitsch, Rasmus Ulslev Pedersen, and Benedikt Huber. Worst-case Execution Time Analysis for a Java Processor. *Software: Prac. and Exp.*, 40(6):507–542, 2010.

[56] Tobias Schoofs, Eric Jenn, Stéphane Leriche, Kelvin Nilsen, Ludovic Gauthier, and Marc Richard-Foy. Use of perc pico in the aida avionics platform. In *Proceedings of the 7th International Workshop on Java Technologies for Real-Time and Embedded Systems*, JTRES '09, pages 169–178, New York, NY, USA, 2009. ACM.

[57] Jesper Smith, Douglas Stephen, Alex Lesman, and Jerry Pratt. Real-time control of humanoid robots using openjdk. In *Proceedings of the 12th International Workshop on Java Technologies for Real-time and Embedded Systems*, JTRES '14, pages 29:29–29:36, New York, NY, USA, 2014. ACM.

[58] JP Smith. *Online swing leg trajectory optimization for a force controllable humanoid robot*. PhD thesis, TU Delft, Delft University of Technology, 2012.

[59] 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 '12, 2012.

[60] Kleanthis Thramboulidis and Alkiviadis Zoupas. Real-time Java in control and automation: a model driven development approach. In *Emerging Technologies and Factory Automation, 2005. ETFA 2005. 10th IEEE Conference on*, volume 1, pages 8–pp. IEEE, 2005.

[61] Reinhard Wilhelm and Daniel Grund. Computation Takes Time, but How Much? *Commun. ACM*, 57(2):94–103, February 2014.

[62] Shuai Zhao. Implementing level 2 of Safety-Critical Java, 2014.