Challenge Benchmarks for Verification of Real-time Programs

Tomas Kalibera[1,2], Pavel Parizek[2], Ghaith Haddad[3], Gary T. Leavens[3], Jan Vitek[1]

[1]Purdue University, [2]Charles University, [3]University of Central Florida

Abstract

Real-time systems, and in particular safety-critical systems, are a rich source of challenges for the program verification community as software errors can have catastrophic consequences. Unfortunately, it is nearly impossible to find representative safety-critical programs in the public domain. This has been significant impediment to research in the field, as it is very difficult to validate new ideas or techniques experimentally. This paper presents open challenges for verification of real-time systems in the context of the Real-time Specification for Java. But, our main contribution is a family of programs, called CD_x, which we present as an open source benchmark for the verification community.

Categories and Subject Descriptors D.2.4 [*Software Engineering*]: Software/Program Verification; C.3 [*Special-Purpose and Application-Based Systems*]: Real-time and embedded systems

General Terms Verification, Experimentation

1. Introduction

Safety-critical systems are real-time systems in which an incorrect response or an incorrectly timed response can result in significant loss to its users; in the most extreme case, loss of life may result from such failures. For this reason, safety-critical applications require an exceedingly rigorous validation and certification process. While traditional approaches to certification, such as the DO-178B certification for airborne systems in U.S. [32], ED-12B in Europe [12], have prescribed software engineering processes and manual verification, there is a growing pressure to use formal, automated, techniques. This is motivated by the rapidly growing size of safety-critical code bases and increasing complexity of modern processors.

The emphasis on timeliness is what sets apart real-time systems from other computer systems. In a hard real-time system, meeting deadlines is just as important as giving correct results. The verification of timeliness is particularly challenging as it requires an understanding of the complete system, down to the hardware. Any change made to a program potentially has consequences on timing due to low-level interactions of hardware, compiler or libraries (by changing cache behavior, processor pipeline states, or memory allocator states). Developers are keenly aware of these interactions and try to reduce them by design. Thus, safety-critical systems are often designed to be verifiable. This may entail restricting the features available on the target platform, for example by limiting the hardware (disabled cache or use of scratch-pad memory [33], reduced or disabled pipelining [24], disabling processors, or even intentionally simplified instructions [25]), the operating system (no virtual memory, no memory protection, simple scheduling algorithms), or the programming model (limited multi-threading, preallocation of system structures, no dynamic memory allocation). Moreover, further reduction of unused code in the operating system and libraries is driven by limited resources on embedded devices (memory, storage, CPU speed and power).

If safety-critical developers are willing to restrict their hardware, operating system and programming methodology, to simplify testing and informal reasoning about their software, it should be possible to get them to adapt their programming to the limitation of automated verification tools. *But this will only happen if these tools can provide developers with useful answers.* Thus the challenge to the programming language and programming verification communities is to come up with languages, methodologies and tools that can scale to real systems and provide the kinds of guarantees that are required for certification.

One particularly vexing challenge for both language and verification communities is the lack of representative, publicly available benchmark programs to evaluate ideas and tools. What is particularly missing, is programs that share the size and complexity characteristics of deployed systems, as well as programs written by domain experts following best-practice software development methodologies. Lacking real-world programs, researchers are left with the choice of using non real-time benchmarks or inventing their own benchmarks. Thus they ended either trying to address problems that do not occur in real-time systems or solve oversimplified abstractions of real programs. It should thus not come as a surprise that very few academic tools have seen practical use (with, of course, some notable exceptions such as AbsInt and Astree).

In this paper we propose challenge problems based on our experience working with ([5, 21, 23, 36]), and implementing ([5, 31]), the Real-time Specification for Java, and we offer a family of representative programs that can be used for validation. The Real-time Specification for Java (RTSJ) [8] is a standard that enhances the Java programming environment with support for writing software systems with real-time characteristics. With multiple commercial offerings RTSJ is becoming a viable alternative for real-time developers [3, 4, 7, 9]. Lockheed Martin used Java to modernize the Aegis cruiser fleet [2], while Raytheon used it for the computing environment software of the DDG-1000 warship [27]. Other applications include unmanned aircraft [5, 18, 19], audio processing [20], industrial automation [15], and flight entertainment [17]. The RTSJ is currently being extended to support certification with the JSR-302 *Safety Critical Java specification* (SCJ) [16]. The SCJ specification is designed to enable the creation of safety-critical applications using a safety-critical Java infrastructure and using safety-critical libraries that are amenable to certification under DO-178B, Level A and other safety-critical standards. In the context of Java

PLPV'10, January 19, 2010, Madrid, Spain.

technology, this means a much tighter and smaller set of Java virtual machines and libraries, and much more precise performance requirements on the virtual machines and libraries. Additionally, the applications must exhibit freedom from exceptions that cannot be successfully handled. This requires, for example, that there be no memory access errors at runtime.

In this paper we focus on certification of source level properties. In a real-world setting, any proof carried out at the source level will have to be traced through the different layers of compilation all the way down to machine code. In the case of many real-time Java VMs this means certifying the translation from source to bytecode, then from bytecode to C, and finally from C to native [3, 4, 31]. This is an important but distinct problem. Furthermore, while analyzing worst case execution time (WCET) is a key for reasoning about timeliness and schedulability, we will not address this issue directly as it requires a deep understanding of the execution platform (operating system and hardware).

The main contributions of this position paper are a list of challenge problems, some well-known others less so, and benchmark programs that can be used to validate solutions to these challenges. For validation we offer CD_x, a family of benchmarks implementing an idealized real-time aircraft collision detection algorithm [21]. While the CD_x application is simple and admittedly idealized, it provides a good starting point for automated verification techniques. In particular, we have made an effort to provide multiple comparable implementations of the benchmark. The x in the name refers to configuration options that let users choose between a plain Java benchmark, an RTSJ or an SCJ version. Furthermore, we offer both (real-time) garbage-collected and region-allocated versions of the code base. Lastly, a version of the software written in C for the RTEMS/LEON embedded platform (being used by both NASA and ESA in space missions [13, 30]) is provided. We also provide earlier versions of the code that contain actual errors.

The benchmarks with instructions how to run them can be downloaded from http://www.ovmj.net/rcd.

2. Real-time Java

The Real-Time Specification for Java was developed within the Java Community Process as the first Java Specification Request (JSR-1). Its goal was 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" [8] 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 (in a single processor application). 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 from a scheduling perspective. Schedulable objects can also have handlers that are released in the event of a deadline miss.
- *Synchronization*: Priority inversion through the use of Java's synchronization mechanism (monitors) is controlled by using the priority inheritance protocol (PIP), or optionally, the priority ceiling emulation protocol (PCEP). This applies to both application code and the virtual machine itself.
- *Memory Management*: Time-critical threads must not be subject to delays caused by garbage collection. To facilitate this, a `NoHeapRealtimeThread` 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*, or an immortal memory area from which objects need never be reclaimed.
- *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 that are marked as allowing asynchronous interruption [10]. This facilitates early termination while preserving the safety of code that does not expect such interruptions.

2.1 Safety Critical Java

The main goal of JSR-302 is to facilitate a certification of Java programs as far as possible. For this purpose, radical subsetting of the full Java environment is required. First of all, as garbage collection is not supported, heap memory is not available, thus some convenient methods of the `java.lang` package can not be supported. Emphasis is placed on using periodic event handlers instead of threads, with preemptive, priority based scheduling. SCJ supports the priority ceiling emulation protocol for avoiding priority inversion, but not the priority inheritance protocol.

The complexity of safety-critical software varies greatly. At one end of the spectrum, safety-critical applications contain only a single thread and support only a single function, with only simple timing constraints. At the other end, there exist highly complex multimodal safety-critical systems. The cost of certification of both the application and the infrastructure is highly sensitive to their complexity, so enabling the construction of simpler applications and infrastructures is highly desirable. Therefore, SCJ defines three compliance levels to which both implementations and applications may conform. The SCJ refers to the distinct compliance configurations, such as Level 0 and Level 1. Level 0 refers to the simplest applications, while Level 1 refers to slightly more complex applications.

2.1.1 Level 0

A Level 0 application's programming model is a familiar model often described as a timeline, frame-based, or cyclic executive model. In this model, a mission can be thought of as a set of computations, each of which is executed periodically in a precise, clock-driven timeline, and processed repetitively throughout the mission. A Level 0 application's schedulable objects shall consist only of a set of Periodic Event Handlers (PEHs). Each PEH has a period, priority, and start time relative to the beginning of a major cycle. A schedule of all PEHs is constructed by either the application designer or by an offline tool provided with the implementation. All PEHs execute under control of a single underlying thread. This enforces the sequentiality of every PEH, so the implementation can safely ignore synchronization. The use of a single thread to run all PEHs without synchronization implies that a Level 0 application runs only on a single CPU. If more than one CPU is present, it is necessary that the state managed by a Level 0 application not be shared by any application running on another CPU. The operations `wait` and `notify` are not available at Level 0 or Level 1. Each PEH has a private scoped memory area created for it before invocation that will be entered and exited at each invocation.

2.1.2 Level 1

A Level 1 application uses a familiar programming model consisting of a single mission with a set of concurrent computations, each with a priority, running under control of a fixed-priority preemptive scheduler. The computation is performed in a set of PEHs and/or Aperiodic Event Handlers (APEHs). A Level 1 application shares objects in mission memory among its PEHs and Aperiodic Event Handlers (APEHs), using synchronized methods to maintain the integrity of its shared objects. Each Level 1 PEH or APEH has a private scoped memory area created for it before invocation that will be entered and exited at each invocation. During execution, the PEH or APEH may create, enter, and exit one or more other scoped memory areas, but these scoped memory areas must not be shared among PEHs or APEHs.

3. Challenge Problems

This section presents a non-exhaustive list of software hazards that have to be prevented and properties that should be established in RTSJ applications. It is not expected that automated techniques alone will be able to establish them for complex real-time systems. Indeed, real-time programmers are used to providing additional information to convince certification authorities. Typically this comes in the from of various design documents, and copious test cases. But it is conceivable that experts could be trained to provide sufficient annotations to automated tools provided that they get, in exchange, strong correctness guarantees.

3.1 Exceptions

The first challenge is to ensure the absence of uncaught exceptions, thrown either by user code, libraries or the virtual machine. The first group of exceptions (see Figure 1) is not specific to real-time Java, but must nevertheless be checked. Out of memory conditions are somewhat special, and will be revisited later. Stack overflow errors require bounding the size of the call stack, but source-level analysis results may be invalidated by low level compiler optimizations which impact the size of individual stack frames (inlining can increase the number of local variables in a frame for instance).

```
ArithmeticException
ArrayIndexOutOfBoundsException
ArrayStoreException
NegativeArraySizeException
NullPointerException
OutOfMemoryError
StackOverflowError
```

Figure 1. Exceptions not specific to RTSJ.

The second group of exceptions (see Figure 2) is specific to real-time Java. The first exception will occur when a schedulable object attempts to lock a `PriorityCeilingEmulation` lock with an effective priority higher than the lock's ceiling. A `MemoryScopeException` occurs when a wait-free queue is referenced from an incompatible memory area (the wait-free end of a queue can only be used from a non-heap region). The next two exceptions are thrown when memory areas are given invalid arguments at construction time. The last four exceptions are thrown in response to invalid memory operations and will be detailed later.

The above focused on uncaught exceptions. Caught exceptions require attention as well. First, while Java allows the specification of catch-all patterns these must be avoided in safety critical systems. Thus automated tools should verify that there is a catch clause for every exception. In general, safety-critical system will strive to avoid try-catch and provide explicit checks instead.

```
CeilingViolationException
MemoryScopeException
SizeOutOfBoundsException
OffsetOutOfBoundsException
IllegalAssignmentError
InaccessibleAreaException
MemoryAccessError
ScopedCycleException
```

Figure 2. Exceptions specific to RTSJ.

3.2 Type Analysis

The size of the code base that must be validated is an important concern in safety-critical applications. It is thus essential to reduce the size of the code base that need to be certified. In object oriented systems, heavy use of features such as subtyping and dynamic binding, while easing software reuse and modularity, make it harder for developers to determine which parts of a system will be exercised in any run. Thus, one of the challenges for tool support is to provide for any call site of the form

```
obj.method(...);
```

a tight bound on the potential class of the object referenced by `obj`. *Thus a second challenge is to tightly bound the size of the set of candidate classes at each dispatch.* It is also noteworthy that in the case of an interface dispatch (i.e. the static type of `obj` is a Java interface) the dispatch is not guaranteed to be constant time. Thus, such operations should either be avoided or shown to have a small fixed-size set of target methods.

Similarly, tools should attempt to predict the outcome of type test expressions (`obj instanceof C`) and checked casts (`(C)obj`).

3.3 Memory Analysis

There are several important questions with respect to memory usage. In a real-time garbage collected real-time system, the key piece of information required to avoid an `OutOfMemoryError` is the allocation rate of each schedulable object. This is needed to ensure that the schedule leaves enough time for the real-time GC to keep up with the application (see [22] for more details). This gives a third challenge: *to automatically determine how many bytes will be allocated per release of each thread or even handler.* For real-time threads, a release is bounded by an invocation of the `waitForNextPeriod()` method, e.g.

```
boolean missedDeadline = false;
while (!missedDeadline) {
  ... // per release operations
  missedDeadline = ! RealtimeThread.waitForNextPeriod();
}
```

In the case of event handlers, the per release cost is based on the memory allocated (transitively) by the event handler's `run()` method. The RTSJ provides a way for declaring the allocation rate of a schedulable object, via the `MemoryParameters` class. If this argument is used, then tools should validate that actual usage is bounded by the value given to the corresponding argument.

For memory allocated in non-heap areas, things are slightly different. Instead of allocation rate, the relevant measure is maximum allocation per activation of the memory area, in a multi-threaded setting an activation ends when all threads that have entered an area exit it. In the case of `ImmortalMemory`, there is a single activation that lasts for the lifetime of the VM. It is necessary to establish that the maximal size of the area is no smaller than sum of the maximum per activation allocations performed by all threads that could potentially execute in it. Each schedulable object may specify its memory usage requirements in terms of `ImmortalMemory` and one associated scoped memory (in the `MemoryParameter` object at-

tached to the schedulable). Tools should validate that these values indeed bound the schedulable's memory usage. While memory allocation mostly occurs via direct invocation of the `new` operation, there are also reflexive construction operations using `newInstance` that require alias analysis to disambiguate the target region, and implicit exceptions (e.g. NullPointerExceptions), which consume memory in the current memory region.

There are other memory related errors that must be prevented. The `ScopedCycleException` and `InaccessibleAreaException` are thrown when a thread performs an invalid enter operation. Preventing these requires modeling the scope stack of each thread.

Finally, in order to prevent dangling pointer errors, the RTSJ VM will throw an `IllegalAssignmentError` on an attempt to store a reference to an object allocated in a shorter lived scope into a field of an object allocated in a longer lived scope. And, for threads that have been marked as no-heap, any attempt to load a reference to an object allocated in the heap will result in a `MemoryAccessError`. To prevent these errors tools will have to resolve for any read or write of a reference variable: where the source and target objects are allocated and what thread is performing the operation.

Scoped memory related errors are not an issue for purely real-time garbage collected systems. In SCJ some errors are ruled out by construction, for instance, `ScopedCycleException` can not occur.

3.4 Blocking Analysis

The fourth challenge is to compute the blocking time of each schedulable object per release, which is critical for schedulability analysis of multi-threaded real-time systems. At the source level this translates to discovering which locks can be contended for among any group of threads and providing a bound on the amount of computation that can be performed within critical section. In general, it is an error to perform a blocking operation within a critical section, thus tools should demonstrate that no I/O or other long latency operation is being performed within a potentially contended lock. If a lock implements a priority ceiling protocol, it must be established that the thread acquiring does not have a higher priority than the ceiling. It is also, in general, an error to call `waitForNextPeriod()` while holding a lock as this may result in unbounded blocking time. Deadlock prevention should be performed although, on a single-processor system, the ceiling protocol can be implemented so that deadlocks are impossible (usually this is done by turning off interrupts during the critical section).

3.5 Loop Bound Analysis

While we do not focus on timing analysis, as this requires more low-level information, one of the necessary inputs for tools such as AbsInt or Rapita is static loop bounds. It is customary for programmers to provides those by hand. Automated tools can help. *The fifth challenge is thus to infer bounds when possible, or at least validate user provided annotations.* There are limits of course, annotations related to the range of inputs (eg. possible values read from external sensors, etc) will have to be trusted.

4. Benchmark Application

We propose a family of benchmarks based on the CD_x suite [21]. CD_x is open source application benchmark suite that targets different hard and soft real-time virtual machines. CD_x is, at its core, a real-time benchmark with a single periodic task, which implements aircraft collision detection based on simulated radar frames. The benchmark can be configured to use different sets of real-time features and comes with a number of workloads. The main components of the application are an *air traffic simulator* and a *collision detector*. The air traffic simulator generates radar frames, each containing a set of aircraft with their current coordinates. The radar frames are generated periodically at a pre-set frequency and are passed to the collision detector. The collision detector maintains a list of the last known aircraft positions. Upon receiving a frame from a radar, the collision detector calculates trajectories from each aircraft (last known stored position to new position received in the radar frame), and checks if any two aircraft are on collision course. The time between two radar frames is very small (i.e. 10ms), and thus the planes can be assumed to travel on straight paths and at constant speed within this time interval. For performance reasons, collision detection is performed in two steps: *2-d reduction* and *3-d collision checking*. The first step is to rule out collisions of aircraft that are very far apart: it ignores aircraft altitude and uses a coarse-grained resolution for latitude and longitude. Only aircraft identified as potentially colliding are checked for collision using the full 3-d collision checker that calculates the minimum distance of two points traveling in time along line segments in 3-d space. The collision detection checker uses dynamic memory allocation, and aircraft positions are stored in a hash table to reduce memory usage. The 3-d collision detection is a task with intensive floating point computation. Both time and memory complexity rise with the number of aircraft and number of collisions (both suspected at some level of the algorithm and finally detected with 3-d collision checking).

The implementation language is Java. The following table summarize the main configuration options that are supported:

$\mathsf{CD}_{\mathtt{jg}ns}$	Plain Java with garbage collection
$\mathsf{CD}_{\mathtt{rs}ns}$	RTSJ with scoped memory
$\mathsf{CD}_{\mathtt{rg}ns}$	RTSJ with real-time garbage collection
$\mathsf{CD}_{\mathtt{ss0}s}$	SCJ with scoped memory

The value of n can be either 0 to indicate the absence computational noise, `j` for the SPEC JVM 98 `javac` benchmark or `s` for the ATS simulator. The value of s defines the ATS implementation: `a` is the ATS simulator, `b` is the version that reads the simulation from a binary file, and `e` is for the case where the simulation is encoded in a Java class.

The plain Java version of CD_x is obtained through wrapper functions that provide plain Java implementations of the requested RTSJ functionality. While the dependency of the benchmark code on RTSJ library can be removed by the wrappers, the impact of RTSJ memory semantics on the architecture could not be abstracted out. The use of scopes and immortal memory by itself requires additional threads in the application. Also, memory assignment rules sometimes lead to the need of copying arguments passed between memory areas (i.e. heap to scope, inner scope to outer scope). Even more, we also structured the code to make it is easier for programmers to keep track of which objects live in which memory areas. Thus, the architecture is representative of an RTSJ application, but not of plain Java application.

The plain Java version of the benchmark can be both compiled and run with standard Java. The RTSJ Java libraries and a RTSJ VM are only needed to build and run the RTSJ version of the benchmark with immortal memory, scopes or RTGC. The RTSJ code has been tested with Sun's Java Real-Time System (RTS), IBM's WebSphere Real-Time (WRT), and Ovm.

To measure the complexity of the benchmark code, we use the Chidamber and Kemerer object-oriented programming (CK) metrics [11] measured with the ckjm software package [34]. We apply the CK metrics to the classes that the application loads. The results are shown in Table 1, separately for the CD and the ATS. The CD only uses selected collection classes from the Java libraries, which we isolated into the `javacp.util` package. For the CD we thus also have the complexity metrics for standard libraries it uses. For the ATS, we exclude the standard libraries from the analysis.

Package Name	WMC	DiT	NOC	CBO	RFC	LCOM	Ce	NPM
Detector								
immortal	35	6	0	21	87	13	8	25
immortal.persistentScope	32	4	0	29	77	6	8	23
immortal.persistentScope.transientScope	196	17	0	41	93	530	58	87
javacp.util	936	113	68	508	1506	7003	474	687
Simulator								
command.*	607	45	53	452	1569	3763	206	611
heap	187	44	18	101	420	511	80	144

WMC	Weighted methods/class	CBO	Object class coupling
DIT	Depth inheritance tree	RFC	Response for a class
NOC	Number of children	LCOM	Lack of method cohesion
Ce	Afferent couplings	NPM	Number of public methods

Table 1. CK metrics for loaded classes.

4.1 Customized Benchmarks

We provide a number of customized versions of CD_x.

We present a benchmark `rtsjmem-error` for methods and tools that aim at checking whether a program does not violate the memory access rules defined by the RTSJ memory model. The code includes a violation of one of the memory access rules that we actually made earlier when refactoring the application. As there are currently no tools for checking of the memory access rules (that we know of), *the challenge is to develop methods and tools (i) that can statically detect violations of the memory access rules and (ii) that scale well to systems at least of size and complexity comparable to the collision detector application.*

We provide three benchmarks for methods and tools aiming at verification of general correctness properties of Java programs — `concur-error`, `charsize-error`, and `error-free`. The `concur-error` benchmark contains a race condition that we discovered using Java PathFinder. Although this error would be discovered by most of the verification tools for Java that aim at concurrency errors, it could be used as a good test for scalability and efficiency of the tools. The challenge is to minimize the time and memory needed to find the error. The `charsize-error` benchmark contains a buffer overflow error that is caused by incorrectly assuming that the size of a character in a native encoding is always one byte. This is not true, for example, in case of Java programs running on platforms that use UTF-8 as the native encoding. Such an error can be discovered only by tools that correctly model size of characters in various encodings and character sets. The `error-free` benchmark does not contain a violation of any non-real-time correctness property, as far as we know. In case of this benchmark, the challenges for verification tools and methods are (i) to traverse the whole state space of the application in reasonable time and memory and show that there are really no errors, or (ii) to find some errors that we are not aware of.

4.2 RTEMS/LEON

The open-source RTEMS operating system [1] and the open-specification of LEON hardware form a platform for embedded real-time systems which is being used by both NASA and ESA in some present missions and is planned for future ESA missions. This platform is thus a realistic target for verification tools. Unlike traditional operating system, RTEMS from the view of an application resembles a library; with RTEMS the build process of an application results in a binary executable on bare hardware. Only the services needed by specific application are included: timer support, individual drivers, individual filesystems, support for a floating point unit, etc. The application programmer has to specify these dependencies, as well as the maximum number of certain OS objects needed (i.e. tasks or semaphores). The system supports multi-tasking without processes (it has no memory protection), and to a limited extent also multi-processing. RTEMS runs on about 20 kinds of processors, including x86 and LEON. The hardware specification of the LEON processor and hardware controllers necessary to build a complete system [14] is available in VHDL, allowing implementation of system-on-chip architectures on FPGA or directly as hardware on ASIC. The processor uses a 32-bit SPARC v8 instruction set with instruction and data caches and with a pipeline. Unlike PRET [25], it is thus not designed to make WCET analysis easy, but to have acceptable performance and to work in space (i.e., with a radiation-hardened RAM). We provide a version of CD_x written in C for the RTEMS/LEON platform. For comparison purposes, we also provide the output of a Java-to-C compiler.

5. Conclusion

There exist many techniques and tools for verification and analysis of Java programs. Some are based on model checking [35], while others are based on static data- and control- flow analysis [6, 28, 29]. However, these techniques and tools have important drawbacks – they do not support all the properties, do not scale to large Java programs, or report a prohibitive number of false positives. The challenge is to improve the precision and scalability of existing methods and tools, or to develop novel techniques, such that the tools can be successfully applied to Real-time Java programs of the size and complexity comparable to the collision detector application. Little work exists that directly address timeliness. While, for instance, there is an extension of Java PathFinder for checking whether a RTSJ program meets all deadlines [26] using discrete event simulation to represent time periods, it does not support any other RTSJ-specific correctness property, and we are not aware of any other work in this area. We hope that making the CD_x benchmark program available, will be enabler for new research.

Acknowledgments. The collision detector benchmark has been originally created by Ben Titzer, then extended by Jeff Hagelberg, Filip Pizlo, and later by Tomas Kalibera. The C version was written by Ghaith Haddad, Petr Maj, and Tomas Kalibera. CK metrics were measured by Ales Plsek. Different versions of the benchmark were used and have been customized at Purdue, IBM, INRIA Lille, and Charles University. This work was partially supported by NSF grants CNS-0938256, CCF-0938255, CCF-0916310 and CCF-0916350, the Grant Agency of the Czech Republic project 201/08/0266, and by the Ministry of Education of the Czech Republic (grant MSM0021620838).

References

[1] Real-time executive for multiprocessor/missile systems (RTEMS). http://www.rtems.com/, 2009.

[2] Aegis. Lockheed Martin selects Aonix PERC Virtual Machine for Aegis Weapon System. *Military Embedded Systems*, 2006. http://www.mil-embedded.com/news/db/?4224.

[3] aicas. The Jamaica Virtual Machine homepage. http://www.aicas.com, 2005.

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

[5] A. Armbruster, J. Baker, A. Cunei, C. Flack, D. Holmes, F. Pizlo, E. Pla, M. Prochazka, and J. Vitek. A real-time java virtual machine with applications in avionics. *ACM Transactions on Embedded Computing Systems (TECS)*, 2007.

[6] C. Artho and A. Biere. Applying static analysis to large-scale, multi-threaded java programs. In *Australian Software Engineering Conference (ASWEC)*, 2001.

[7] J. Auerbach, D. F. 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 *ACM & IEEE International Conference on Embedded Software (EMSOFT)*, 2007.

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

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

[10] B. Brosgol, S. Robbins, and R. Hassan II. Asynchronous transfer of control in the Real-Time Specification for Java. In *International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC)*, 2002.

[11] S. R. Chidamber and C. F. Kemerer. A metrics suite for object oriented design. *IEEE Transactions on Software Engineering*, 1994.

[12] EUROCAE. EUROCAE ED-12B software considerations in airborne systems and equipment certification, 1992.

[13] European Space Agency. Venus express mission. http://www.esa.int/SPECIALS/Venus_Express, 2009.

[14] J. Gaisler, E. Catovic, M. Isomki, K. Glembo, and S. Habinc. GRLIB IP core users manual. http://www.gaisler.com/products/grlib/grip.pdf, 2009.

[15] S. Gestegard Robertz, R. Henriksson, K. Nilsson, A. Blomdell, and I. Tarasov. Using real-time Java for industrial robot control. In *International Workshop on Java Technologies for Real-time and Embedded Systems (JTRES)*, 2007.

[16] 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.

[17] Inflight. Aonix PERC selected for inflight entertainment system. *Embedded Computing Design*, 2007. http://www.embedded-computing.com/news/db/?8205.

[18] J-UCAS. Boeing selects software for J-UCAS X-45C. *Defense Industry Daily*, 2005. http://www.defenseindustrydail.com/boeing-selects-software-for-jucas-x45c-01413/.

[19] JB. The JamaicaVM brings Java technology to mission software in an unmanned aircraft by EADS. *Military Embedded Systems*, 2006. http://www.mil-embedded.com/news/db/?3302.

[20] N. Juillerat, S. Müller Arisona, and S. Schubiger-Banz. Real-time, low latency audio processing in Java. In *International Computer Music Conference (ICMC)*, 2007.

[21] T. Kalibera, J. Hagelberg, F. Pizlo, A. Plsek, B. Titzer, and J. Vitek. CDx: A family of real-time Java benchmarks. In *International Workshop on Java Technologies for Real-time and Embedded Systems (JTRES)*, 2009.

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

[23] T. Kalibera, M. Prochazka, F. Pizlo, J. Vitek, M. Zulianello, and M. Decky. Real-time Java in space: Potential benefits and open challenges. In *Proceedings of DAta Systems In Aerospace (DASIA)*, 2009.

[24] E. Lee and D. Messerschmitt. Pipeline interleaved programmable dsp's: Architecture. *IEEE Transactions on Acoustics, Speech and Signal Processing*, 1987.

[25] B. Lickly, I. Liu, S. Kim, H. D. Patel, S. A. Edwards, and E. A. Lee. Predictable programming on a precision timed architecture. In *International conference on Compilers, Architectures and Synthesis for Embedded Systems (CASES)*, 2008.

[26] G. Lindstrom, P. C. Mehlitz, and W. Visser. Model checking real time Java using Java PathFinder. In *Automated Technology for Verification and Analysis (ATVA)*, 2005.

[27] B. McCloskey, D. Bacon, P. Cheng, and D. Grove. Staccato: A parallel and concurrent real-time compacting garbage collector for multiprocessors. Research report, IBM, 2008. http://www.eecs.berkeley.edu/~billm/rc24504.pdf.

[28] M. Naik, A. Aiken, and J. Whaley. Effective static race detection for java. In *Programming Language Design and Implementation (PLDI)*, 2006.

[29] M. Naik, C.-S. Park, K. Sen, and D. Gay. Effective static deadlock detection. In *International Conference on Software Engineering (ICSE)*, 2009.

[30] National Aeronautics and Space Administration. Dawn mission. http://dawn.jpl.nasa.gov, 2009.

[31] F. Pizlo, L. Ziarek, and J. Vitek. Towards Java on bare metal with the Fiji VM. In *International Workshop on Java Technologies for Real-time and Embedded Systems (JTRES)*, 2009.

[32] RTCA and EUROCAE. Software considerations in airborne systems and equipment certification. *Radio Technical Commission for Aeronautics (RTCA), European Organization for Civil Aviation Electronics (EUROCAE), DO178-B*, 1992.

[33] N. R. Shah. Memory issues in PRET machines. Technical report, Columbia University, 2008.

[34] D. D. Spinellis. CKJM Chidamber and Kemerer metrics software, v 1.6. *Technical report, Athens University of Economics and Business*, 2005. http://spinellis.gr/sw/ckjm.

[35] W. Visser, K. Havelund, G. P. Brat, S. Park, and F. Lerda. Model checking programs. *Automated Software Engineering*, 2003.

[36] L. Zhao, D. Tang, and J. Vitek. A technology compatibility kit for safety critical Java. In *International Workshop on Java Technologies for Real-time and Embedded Systems (JTRES)*, 2009.