Model Checking Real Time Java Using Java PathFinder

Gary Lindstrom*, Peter C. Mehlitz+, and Willem Visser+

*University of Utah, +NASA Ames Research Center

Abstract

The *Real Time Specification for Java* (RTSJ) is an augmentation of Java for real time applications of various degrees of hardness. The central features of RTSJ are real time threads; user defined schedulers; asynchronous events, handlers, and control transfers; a priority inheritance based default scheduler; non-heap memory areas such as *immortal* and *scoped*, and non-heap real time threads whose execution is not impeded by garbage collection. The Robust Software Systems group at NASA Ames Research Center has JAVA PATHFINDER (JPF) under development, a Java model checker. JPF at its core is a state exploring JVM which can examine alternative paths in a Java program (e.g., via backtracking) by trying all nondeterministic choices, including thread scheduling order. This paper describes our implementation of an RTSJ profile (subset) in JPF, including requirements, design decisions, and current implementation status. Two examples are analyzed: jobs on a multiprogramming operating system, and a complex resource contention example involving autonomous vehicles crossing an intersection. The utility of JPF in finding logic and timing errors is illustrated, and the remaining challenges in supporting all of RTSJ are assessed.

1 Overview

The possibility of using Real Time Specification for Java (RTSJ) [fJEG] software on future missions is under consideration at NASA, for all the familiar reasons: standardized (i.e., platform independent) semantics, a rich and vigorous marketplace of implementations and tools, and the overall software engineering advantages of Java as a type safe object-oriented programming language. RTSJ is not based on any Java core language extensions; rather, all its capabilities are conveyed by new classes with special semantics, albeit with some refinement of semantics for existing Java classes. This design decision in effect strikes a bargain: less run time

predictability, in exchange for language stability. An alternative choice might have been to enhance the declarative content of the language in the interest of stronger compile time program validation, as was done for example with exceptions in Java.

The dual consequence of this design decision is inadequacy of static analysis for RTSJ software verification and validation, and a corresponding vital need for techniques performing dynamic analysis, e.g., model checking. We report here on an application of the JAVA PATHFINDER model checker (JPF) [VHB+03,JPFa] to RTSJ programs, focusing on the latter's dynamic, time quantified behavior, with the goal of developing a tool capable of validating RTSJ applications, ideally to the level of mission deployability. Our approach emphasizes the central issue of temporal correctness (e.g., threads meeting deadlines) under nondeterministic choices; correctness of memory usages and asynchronous control flow are reserved for future work. Thus we are focusing on *classical* correctness issues in real time software, rather than issues related to specialized JVM behavior.

Our approach uses discrete state simulation (DSS) as a basis for modeling time. Real time threads are modeled as ordinary Java threads, constrained to run one at a time, i.e., as *coroutine*'s. Their interactions, e.g., through CPU scheduling, are modeled by resource contention techniques familiar to DSS programming (a summary of DSS concepts is given in §3). This permits execution of programs within our RTSJ profile on any Java implementation.

However, two important capabilities are provided by analyzing (running) RTSJ programs under JPF: (a) execution cost logging at the bytecode level, and (b) alternative execution path exploration via nondeterministic choice selection. Point (a) permits closing an important causality loop impossible on an ordinary JVM:

$$\textit{thread execution cost} \rightarrow \textit{deadline misses} \rightarrow \textit{miss events} \rightarrow \textit{event handlers} \rightarrow \textit{additional thread execution cost}$$

Analyzing such loops is a critical requirement in the validation and verification of complex RTSJ applications, and is well beyond the capability of current static analyzers.

2 RTSJ Under JPF: Requirements and Objectives

The first question is clearly *what does it mean to model check an RTSJ program?* The starting point is to view the RTSJ program as just another

Java program (albeit with a class library with special semantics), and simply execute it using the model checking vigilance of JPF. This is fine, except that this presumes the availability of an RTSJ enabled JVM within JPF, which we do not have.

Unlike a simple Java program, in which the notion of time generally plays an insignificant role, time in RTSJ programs plays a major correctness role, e.g., in quantifying real time deadlines. Moreover, an RTSJ program (the *embedded* program) must be exercised within an implementation of its environment (the *embedding* program). In our view, specifying and constructing such environments are often tasks of difficulty equal to or greater than that of the embedded system. An example is a flight control system, where a fully accurate embedding system must model all the dynamics of the aircraft, as is done in a flight simulator. Hence ensuring that embedding code is correct is as important (or more so) than ensuring that the embedded code is correct.

We adopted the following goals for model checking RTSJ under JPF:

1. Make no changes to the JPF implementation – clearly, a major software engineering win if achievable.
2. Implement the embedding code in Java, and model check the entire combined system – a major validation win if possible.
3. Deal with time through DSS modeling – a familiar and well understood technology.
4. Implement all RTSJ thread interactions (e.g., priority based scheduling with priority inversion avoidance via priority inheritance) through resource contention techniques traditional to DSS.
5. Exploit the run time cost accounting capabilities of JPF to detect deadline misses by real time threads, and to take appropriate actions, e.g., invoking overrun handlers in the embedded code.
6. Finally, utilize the path coverage capabilities of JPF to locate bugs involving nondeterminacy and race conditions (e.g., trying all possible orders of events scheduled at equal simulation times, thereby ensuring the absence of *instant splitting* errors in the code). An important additional benefit is the utility of nondeterministic choice points in the embedding code in obtaining greater test coverage.

3 Step 1: RTSJ in a Simulation Environment

The first step in model checking RTSJ is to implement a profile of RTSJ as a set of conventional Java classes. This we have done to a first level of realism – several features have yet to be implemented, as discussed in §10.

The classes in our implementation include RealtimeThread, PriorityScheduler, AsyncEvent, AsyncEventHandler, OneShotTimer, PeriodicTimer and RelativeTime.

The fundamental concepts of DSS (as developed in the Simula system of the 1970's [BDMN73]) can be summarized as follows:

- Individual *processes* (the traditional terminology – henceforth we will use *thread*) are conceptually concurrent, but in fact execute in an interleaved fashion as coroutines, as mentioned above.
- A thread may be *executing*, *activated*, or *passivated*.
  - An *executing* thread is the one currently running as a coroutine;
  - An *activated* thread is not executing, but is scheduled to do so in the future at a time indicated its event notice on the simulation's event list.
  - A *passivated* thread is neither executing nor active; such threads are typically waiting for some condition to become true, such as being granted a resource.
- Scheduling operations on threads include *activate* (schedule), *passivate*, and *hold*, which is a compound operation comprising activation at a later scheduled time time, and passivation.
- The main thread controls the overall simulation by repeatedly dequeueing from the event list the event notice with the earliest event time, advancing the simulation clock to the time in that event notice, and notifying the associated thread to run – until the event list becomes empty, or a global shutdown operation is invoked.

Since RealtimeThread's are constrained to run as coroutines, the JVM scheduler has only one scheduling choice possible, and DSS event based scheduling is used in an *outboard* manner to orchestrate thread interleaving. For example, Fig. 1 gives our implementation of hold(RelativeTime t).

As mentioned in §2, all RealtimeThread interactions are achieved by contention for Resource objects, e.g., a *CPU*. The upshot is that no changes are necessary to the schedulers of the underlying JVM or JPF to implement scheduling policies such as priority inheritance with FIFO ordering within priorities, as required by the default RTSJ scheduler. Since Java's real time clock is replaced by the simulation clock, all RTSJ executions in this implementation are deterministic (repeatable), even if they use pseudo random methods to draw numbers from probability distributions (assuming fixed seeds) or offer the option of pseudo randomly selecting orders of events scheduled at identical times.

```
public static void hold( RelativeTime t ) throws InterruptedException {
  RealtimeThread currentThread = (RealtimeThread)Thread.currentThread();
  synchronized ( currentThread ) {
    // schedule this thread to run again after hold period
    activate( currentThread, clock.getTime().add(t) );

    // signal main thread to perform next event in simulation cycle
    currentThread.notify();

    // wait for hold to be over
    currentThread.wait();
  }
}
```

Fig. 1. The implementation of hold(t).

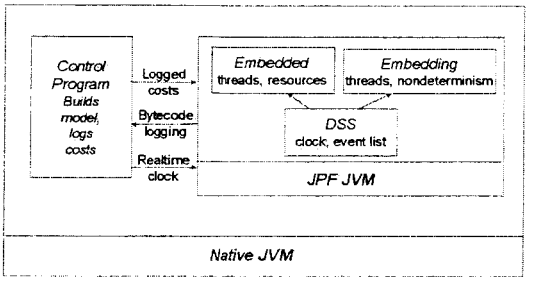

Fig. 2. RTSJ architecture under JPF.

4 Step 2: Combining RTSJ and JPF

Embedded code written in our RTSJ profile, together with its embedding test code using DSS facilities including simulated time, comprise an ordinary Java program that can be run under any Java implementation (without accurate run time modeling, however). The next step is to run the combined program under JPF, with the following additional benefits:

- *Nondeterministic state exploration*, including all orderings of events scheduled for the same instant, and choice points in the embedding code, and
- *Cost accounting*, with overrun detection and invocation of appropriate handlers, as described below.

Our adaptation of JPF is being done in two stages. The first stage exploits two customization features already available in JPF: its JVM

listener interface [JPFb], and its *Model Java Interface* (MJI) [JPFc] (both features are utilized in the *Control Program* box in Fig. 2).

- *JVM listener interface:* Logging run time (albeit idealized) for Java code under JPF can be done using JPF's JVM listener interface, which invokes control program listener methods on various occurrences, including the execution of each byte code instruction. We use a very simple accounting technique here, whereby each byte code is assigned a fixed run time in a look up table. By this technique the execution time (summed byte code costs) from the start to the end of a RealtimeThread can be accumulated. Similarly, this interface is used to detect execution path backtracking by the JPF JVM, so that path specific accounting data structures can be correspondingly backtracked.
- *Model Java interface:* The MJI interface permits Java code executing under JPF's specialized JVM to access the underlying JVM for access to native facilities. This turns out to be crucial in arranging that run time cost logging, which executes *outside* the JPF JVM, is accessible to the RTSJ application code, which executes *within* the JPF JVM. For example, suppose an AsyncEventHandler invocation has a run time in excess of its stipulated limit, as observed through an MJI native method. This can trigger the invocation of an *overrun event handler*, which must execute within the JPF JVM.

The second and more difficult stage of adapting JPF for RTSJ concerns features that must be implemented by JVM modifications. These features, which include non-heap memory areas and non-heap real time threads, as well as asynchronous control transfers, are discussed in §7.1.

5 Scheduling Policies

We now give more details on our control of scheduling by means of resource contention policies. We illustrate our approach by discussion of five representative policies: *FIFO*, *priority*, *priority inheritance*, *priority ceiling*, and *preemption*. The first two are naive policies inviting priority inversion; the third is obligatory in RTSJ's default scheduler; the fourth is an explicit option, and the RTSJ specification is silent on the fifth.

FIFO: This simplistic policy guarantees fairness, but ignores thread priority.

Priority: Here threads waiting for a resource are selected by (fixed) priority first, and then by FIFO within equal priorities. This policy, as well as FIFO above, provides no defense against priority inversion.

Priority inheritance (PI): This well known policy works by increasing the priority of the thread possessing a PI resource to equal the maximum priority of any thread waiting for that resource (its *dynamic priority*). There are two perhaps unobvious consequences of this policy:

1. Since a thread may possess multiple resources, its dynamic priority is based on the maximum priority of any thread waiting for any of the resources it possesses, and
2. The priorities involved are of course dynamic priorities, so an attempted seize of a resource held by a thread waiting for another resource can cause cascaded priority inheritance effects (and conversely for release's).

Priority ceiling (PC): A PC resource has a fixed priority (its *ceiling priority*) which is used to temporarily elevate the priority of any thread possessing it. If a thread has a dynamic priority greater than the resource's ceiling priority, an attempt to seize the resource causes a PriorityCeilingException to be thrown (the absence of which is an important verification condition).

Preemption: A resource managed under this policy does not change a thread's priority when seized. A thread seizing a resource of this kind only waits if the resource is currently held, and the thread's priority is less than or equal to the priority of the thread holding the resource. If the thread's priority is greater that that of the thread holding the resource, it *steals* the resource.

Modeling the first four policies is straightforward DSS programming. The fifth policy, *preemption*, is a bit trickier, because possession periods (e.g., modeling computational activity by a thread using a CPU resource) can be prematurely ended when the resource is stolen by a higher priority thread. This can be implemented by wrapping such hold method calls in loops that sum actual hold times, and re-exert hold invocations until the stipulated hold time is attained. All five policy implementations easily generalize to multiprocessing systems by managing pools of CPU resources.

6 Applications

We now discuss application of our RTSJ implementation in JPF to two example programs. The first is a simple model of a multiprogramming operating system (OS), while the second is a complex resource contention example involving autonomous cars crossing an intersection. The utility of JPF in finding logic and timing errors in each is illustrated.

6.1 Multiprogramming Operating System

This example models a simple multiprogramming computer system, where jobs (as RealtimeThread's) contend for a CPU, which is a resource of one of the five types discussed in §5. Of these, *preemption* is the most interesting, because (i) it guarantees absence of priority inversion, (ii) it is pervasive in modern operating systems, (iii) its behavior on realistic job mixes defies static analysis, and consequently (iv) real time OS's typically do not employ it, despite the appeal of (i).

CPU	*Job1 (6)*	*Job2 (5)*	*Job3 (4)*	*Job4 (3)*	*Time*
FIFO	3681 / 72% / 6.0	3780 / 73% / 5.0	3879 / 74% / 4.0	3979 / 74% / 3.0	3979
Priority	1891 / 46% / 6.0	1990 / 49% / 5.0	3880 / 74% / 4.0	3979 / 74% / 3.0	3979
PC (6)	1891 / 46% / 6.0	1990 / 49% / 5.2	3880 / 74% / 4.5	3979 / 74% / 3.7	3979
PI	1891 / 46% / 6.0	1990 / 49% / 5.2	3880 / 74% / 4.0	3979 / 74% / 3.2	3979
Preempt	1004 / 0% / 6.0	2008 / 49% / 5.0	3012 / 66% / 4.0	4015 / 74% / 3.0	4015

Fig. 3. Multiprogramming results, by resource type. The parenthesized number in for each job indicates its priority. Columns for each job indicate its duration in simulated milliseconds, followed by its percentage wait time and average priority. *Time* is the completion time of the entire job mix, in simulated milliseconds.

A fixed job mix was analyzed using our RTSJ implementation in JFP, using CPU's of each of our five resource types. The results are given in Fig. 3. In this scenario, there are four jobs that are identical in behavior (10 compute / wait cycles), with identical wait times between cycles. They are all started at time zero. This simple *stress test* keeps the CPU 99% busy independent of its resource type (the simulation ends after the last job terminates). The following observations can be made of the results in Fig. 3:

- The *FIFO* CPU gives the most fair service to the four jobs – because it ignores priority.
- The *Priority*, *Priority Ceiling*, and *Priority Inheritance* CPUs deliver identical service, because the priority of a job only affects its competitive position when more than one job is waiting for the CPU, which does not occur in this simple scenario (an example of priority improving service is given in §6.2).
- Jobs under the *Preemptable* CPU finish strictly according to priority. However, the overall completion time is slightly longer, due to the additional scheduling overhead.

When run under JPF with nondeterminism turned on, there are $4! = 24$ choices for activation order at time zero for the four jobs (the statistically rare case of events scheduled at exactly the same time does not occur after simulation start). Priority inversion was detected in all 24 paths under *FIFO* and *Priority* CPUs, and on no paths under *Priority Ceiling (6)*, *Priority Inheritance*, and *Preemptable* CPUs.

6.2 Intersection Crossing

The example in §6.1 emphasizes the effect of role of resource types in thread scheduling. Our second application is a more complex example, illustrating more advanced features of our RTSJ implementation in JPF. This models autonomous cars transiting an intersection, where the cars (real time threads) can drive straight through, turn right, or turn left. Cars are given priorities chosen from 1 to 8.

The intersection is modeled by four sectors (*NW*, *NE*, *SW*, *SE*), each of which is a resource. For a car driving north, turning right requires possession of sector *SE*; driving straight requires *SE* and *NE* (granted simultaneously, to avoid deadlock; *SE* is released half way through), and turning left involves (i) seizing *SE* and *NE* together; (ii) releasing *SE*, (iii) seizing *NW*, (iv) releasing *NE* and *NW*. The net effect is a model of an uncontrolled intersection of two lane roads, where cars follow the common conventions that a car can drive straight through if the car on its left (if any) is not driving straight through or turning left, the car on its right (if any) is not driving either straight, left or right, and the opposing car (if any) is not turning left.

These rules are complex but deadlock free, which as been confirmed (for specific scenarios) by exhaustive search using JPF on initial event scheduling orders. By comparison, deadlocks caused by the naive policy of seizing all of *SE*, *NE*, and *NW* for a northbound car making a left turn (and correspondingly for cars heading in other directions) were quickly located by JPF.

Car speed is governed by car priority, in the following manner. The time required by a car to transit a sector is $t = 100\ sec/p$, where p is the car's priority. At the extremes, $p = 1$ yields a sector transit time of 100 seconds, and $p = 8$ yields 12.5 seconds. Experiments were run using four resource types for sectors: *FIFO*, *priority*, *priority ceiling* 8, and *priority inheritance*. There are ready intuitions for each of these cases: *FIFO* is round robin, *priority* is fastest vehicle first, *priority ceiling* is a minimum sector speed, and *priority inheritance* is when one sees an ambulance

Sector type	*Car 0 (5/N/S)*	*Car 1 (2/S/L)*	*Car 2 (8/E/L)*
FIFO	33 / 0%	183 / 18%	49 / 21%
Priority	33 / 0%	183 / 18%	49 / 21%
PC(8)	25 / 0%	63 / 40%	45 / 15%
PI	30 / 0%	180 / 16%	46 / 17%

Fig. 4. Intersection results using four sector resource types. (5/N/S) indicates that *Car 0* has priority 5, is heading north, and going straight, etc. *Car 0* and *Car 1* start at time 0; *Car 2* has a start delay of 5 seconds. The figures in each column are completion time in seconds, and percentage wait time.

rapidly approaching, and speeds up accordingly. The *preemption* case is physically impossible!

Sample results are shown in Fig. 4. Note that all cars benefit from higher priority under *priority ceiling*, and marginally so under *priority inheritance*.

The utility of run time cost logging under JPF was demonstrated by giving each car a maximum lifetime (its *release deadline* in RTSJ's vocabulary). If the deadline is set uniformly at 75 seconds, under *priority inheritance* the RTSJ *miss handler* for *Car 1* is invoked, but not for *Cars 0* or *Car 2*.

The above analysis can be accomplished under both native Java and JPF, since it is based solely on simulated time. By contrast, analysis of *miss handler* behavior in RTSJ programs can only be exercised under JPF, where a listener method in our control program records each byte code execution in the subject program. To demonstrate this capability, an onboard computer was postulated for each car (its *autonomous* controller), and a *cycle soaker* method was invoked during passage through each sector (arbitrarily set at 100,000 double divides, with 100 nanosecond cost per byte code; a total of 1,400,024 DDIV's are observed in the deterministic case). If a cost limit of 350 milliseconds is imposed, under *priority inheritance Car 0* terminates without handler invocation, *Car 2* terminates with cost overrun handler invocation, and *Car 1* terminates with both handlers invoked.

7 Critique of JPF

This application breaks new ground for JAVA PATHFINDER in its focus on quantified time as a program correctness issue. Much as been learned about its flexibility in supporting this new and unanticipated correctness

dimension, as well as the limits of our approach that implements RTSJ without making any modifications to JPF.

7.1 Features Not Easily Implemented Under This Approach

In §4 we indicated two areas pose more difficult challenges, which we believe can only be implemented by JVM modification:

- ScopedMemoryArea's and NoHeapRealtimeThread's, which deal with non garbage collected MemoryArea's, and
- Asynchronous transfers of control (ATC), e.g., threads that implement the Interruptible interface and methods that throw AsynchronouslyInterruptedException.

While it may be possible in principle to implement at least the first these features using per-bytecode analysis in a JPF listener method, the overhead of this approach is likely to be prohibitive.

7.2 Opportunities For Application of Other JPF Features

This project thus far has used only basic JAVA PATHFINDER features. Several advanced features of JPF offer attractive opportunities for increased utility in verifying RTSJ programs.

Heuristic search: The default program path exploration strategy is depth first search, using backtracking. Other strategies, such as bounded breadth-first search, can selectively search longer paths due to elimination of the backtrack stack [GV04]. Several criteria for preferring paths in RTSJ programs with higher error potential are evident, such as favoring states with threads whose extrapolated completion time is beyond their stipulated deadlines.

State abstraction: By default JPF saves all previously encountered program states and performs precise equality checks to detect re-encountered states. This policy has several consequences, including (i) significant space overhead, and (ii) inability to recognize states that insignificantly vary from previously seen states. In particular, the extremely fine representation of time in RTSJ (to nanosecond precision), exacerbates (ii). To illustrate, consider state abstraction methods focusing on the core data structure of our system, the scheduled event list. Opportunities for abstraction here include *fuzz* on scheduled event times, e.g., equality to resolution of say 100 nanoseconds, or even ignoring event times altogether, and considering two event lists to be equal if they reference the same real

time threads positioned at the same execution point (say, method and byte code address).

Symbolic execution: JPF interfaces to a constraint system that can solve equations involving linear inequalities [SKV03]. This presents the possibility of asserting constraints on scheduled event times.

- For example, it could be asserted that event e_1 should run at time $t_0 + t(e_2)$, where $t(e)$ is the scheduled time of an event e, and $t(e_2)$ is not yet known, i.e., is symbolic. When $t(e_2)$ becomes bound, e_1 would be scheduled at a concrete time.
- Now suppose two scheduled events e_1 and e_2 have symbolic event times $t(e_1)$ and $t(e_2)$, and the event list is otherwise empty. We then have two options to pursue nondeterministically: (a) e_1 runs next, $t(e_1) \leq t(e_2)$ is asserted, and the simulation clock is set (symbolically) to $t(e_1)$, or (b) symmetrically, e_2 runs next, $t(e_2) \leq t(e_1)$ is asserted, and the simulation clock is set to $t(e_2)$.

Fault driven automatic test case generation: The execution driven symbolic constraint refinement technique just sketched can be the basis for finding necessary and sufficient conditions that lead to specific faults [VPK04]. For example, suppose the real time code is modeling the performance of an aircraft pre-landing checklist. There have been published accident scenarios where a mandatory aircraft response, e.g., completion of landing gear deployment, did not occur in time to ensure the safety of the next step in the checklist, and the pilot under time pressure (the ground is approaching) inappropriately proceeded [Deg04]. Conditions revealing such flaws in real time checklist procedures might be determined by symbolic execution in this manner.

8 Performance

We now present performance figures for our RTSJ profile implementation in JPF. All performance figures are taken from executions in the Eclipse Java IDE with a heap size of one gigabyte on a Pentium 2 laptop with 768MB of RAM.

Our system can be run in five modes: native Java with deterministic or pseudo random choice selection, or JPF with deterministic, pseudo random, or nondeterministic choice selection. We have tested our system in all five modes on the applications presented in §6. Run time figures for the multiprocessing operating system example in § 6.1 under deterministic mode are 120ms for native Java vs. 6,257ms under JPF (the pseudo

random mode numbers are analogous). These absolute numbers are not important; instead, their relative magnitudes are more informative. Two observations emerge: (a) the native Java implementation is quite fast, and (b) the JPF implementation is slower by a factor of about 50 – but it must be remembered that under JPF an interpretive JVM (written in Java) is being employed, cost logging presents a linear execution time overhead, and state saving is performed to support exploration of alternative execution paths (not exploited in the deterministic and pseudo random cases).

CPU type	Run time
FIFO	79.4 sec
Priority	80.9 sec
PC(6)	91.6 sec
PI	99.6 sec
Preemptable	106.2 sec

Fig. 5. Run times for the multiprogramming example under JPF nondeterministic search (backtracking over 24 paths).

To illustrate the cost of JPF state exploration, the CPU example was run under nondeterminism, exploring the $4! = 24$ choices for activation order at time zero for the four jobs discussed in § 6.1 Results are shown in Fig. 5.

9 Related Work

Model checking of timed automata representations has become very popular (see [BY04] for a good overview) for the analysis of real time systems. Our approach differs in that it uses a model checker to analyze RTSJ programs (a richer notation), but we only check safety properties (e.g., classic Java errors such as uncaught exceptions and assertion violations as well as RTSJ properties such as priority inversion).

It has been reported that more than 3000 people have used the RTSJ reference implementation or a commercial RTSJ-compliant JVM to create application prototypes [Loc04]. Tools are available to benchmark RTSJ implementations [CS02].

Model checking is a vigorously evolving research area. Examples of model checking applied to Java programs are Bandera [Ban], Bogor [DHHR05], and the work of Bart Jacobs et al. on *JavaCard* verification [JMR04]. A

closely related area is run time verification of Java systems [KKLS01]. Capability for dealing with time in model checkers has also been evolving rapidly, often through monitoring of event sequences with respect to assertions in linear time logic (LTL) [Hav]. RTSJ itself is drawing critical and insightful analysis, such as the work on Ravenscar [Bur,Wel04].

10 Status and Continuing Work

Our implementation of RTSJ within a DSS environment is operational. RTSJ features supported include RealtimeThread's, AsyncEvent's and AsyncEventHandler's, cost overrun handlers, binding of external happenings to events, simulated and real time Clock's, and various timers, e.g., OneShotTimer and PeriodicTimer, and PhysicalMemoryArea's. API documentation including designation of individual classes and methods not implemented is publicly available [Lin].

Continuing work includes:

1. Maximizing the RTSJ profile we can implement without JVM modification,
2. Development of a more realistic, calibrated execution cost model, taking into account effects of garbage collection, JIT compilation, class loading, etc.,
3. Development of more challenging test cases, with assessment of the scalability of RTSJ under JPF,
4. Extending JPF's JVM (written Java) to include the remaining crucial RTSJ features summarized in § 7.1 (probably using Ravenscar's profile as a guide), and
5. Exploiting advanced JPF features to increase the scale of RTSJ systems that can be analyzed, through techniques such as search heuristics, state abstraction and symbolic constraint analysis [PPV05] as discussed in § 7.2.

Acknowledgements

Michael R. Lowry conceived this project and is providing the resources. The critical comments of Robert E. Filman are gratefully acknowledged.

References

[Ban] http://bandera.projects.cis.ksu.edu/.

[BDMN73] G. M. Birtwistle, O.-J. Dahl, B Myhrhaug, and K. Nygaard. *Simula BEGIN*. Auerbach/Studentliteratur, Philadelphia, 1973.

[Bur] Alan Burns. The Ravenscar profile. http://polaris.dit.upm.es/~ork/documents/RP_spec.pdf.

[BY04] Johan Bengtsson and Wang Yi. Timed automata: Semantics, algorithms and tools. In W. Reisig and G. Rozenberg, editors, *Lecture Notes on Concurrency and Petri Nets*. Springer-Verlag, 2004. LNCS 3098.

[CS02] Angelo Corsaro and Douglas C. Schmidt. Evaluating real-time java features and performance for real-time embedded systems. In *Proc. 8th Real-Time and Embedded Technology and Applications Symposium*. IEEE Computer Society, September 24-27, 2002.

[Deg04] Asaf Degani. *Taming HAL: Designing Interfaces Beyond 2001*. Palgrave Macmillan, 2004.

[DHHR05] Matthew B. Dwyer, John Hatcliff, Matthew Hoosier, and Robby. Building your own model checker using the bogor extensible model checking framework. In *In Proc. 17th Conference on Computer-Aided Verification (CAV 2005)*, 2005.

[fJEG] The Real-Time for Java^{TM} Expert Group. https://rtsj.dev.java.net.

[GV04] A. Groce and W. Visser. Heuristics for model checking Java programs. *International Journal on Software Tools for Technology Transfer*, 2004.

[Hav] Klaus Havelund. Eagle Flier, a rule-based runtime verification framework. http://yangtze.cs.uiuc.edu/~ksen/eagle/.

[JMR04] B. Jacobs, C. Marche, and N. Rauch. Formal verification of a commercial smart card applet with multiple tools. In C. Rattray, S. Maharaj, and C. Shankland, editors, *Algebraic Methodology and Software Technology (AMAST'04)*, pages 21–22. Springer LNCS 3116 2004.

[JPFa] http://javapathfinder.sourceforge.net/.

[JPFb] http://ase.arc.nasa.gov/jpf/Listeners.html.

[JPFc] MJI – the Model Java Interface, http://ase.arc.nasa.gov/jpf/MJI.html.

[KKLS01] Moonjoo Kim, Sampath Kannan, Insup Lee, and Oleg Sokolsky. JavaMaC: a run-time assurance tool for Java. In *First International Workshop on Run-time Verification*. Paris, France, July 23, 2001. Electronic Notes in Theoretical Computer Science, vol. 55 No. 2.

[Lin] Gary Lindstrom. RTSJ-JPF API. http://www.cs.utah.edu/~gary/RTSJ/doc/.

[Loc04] C. Douglass Locke. Real-time java moving into the mainstream. *RTC Journal*, January 2004.

[PPV05] Corina Pasareanu, Radek Pelanek, and Willem Visser. Concrete model checking with abstract matching and refinement. In *In Proc. 17th Conference on Computer-Aided Verification (CAV 2005)*, 2005.

[SKV03] C. S. Parareanu S. Khurshid and W. Visser. Generalized symbolic execution for model checking and testing. In *Proceedings of TACAS*, April 2003.

[VHB+03] W. Visser, K. Havelund, G. Brat, S. Park, and F. Lerda. Model checking programs. *Automated Software Engineering Journal*, 10(2), April 2003.

[VPK04] Willem Visser, Corina S. Pasareanu, and Sarfraz Khurshid. Test input generation with Java PathFinder. In *Proceedings of ISSTA*, July 2004.

[Wel04] Andy Wellings. *Concurrent and Real-Time Programming in Java*. John Wiley & Sons, Ltd., Chichester, West Sussex, England, 2004.