A Simulation Based Model Checker for Real Time Java

Gary Lindstrom
School of Computing
University of Utah
Salt Lake City, UT 84112-9205 USA
gary@cs.utah.edu

Peter C. Mehlitz
NASA Ames Research Center
Mail Code 269-2
Moffett Field, CA 94035-1000 USA
pcmehlitz@email.arc.nasa.gov

Willem Visser
NASA Ames Research Center
Mail Code 269-2
Moffett Field, CA 94035-1000 USA
wvisser@email.arc.nasa.gov

ABSTRACT

The *Real Time Specification for Java* (RTSJ) is an augmentation of Java for real time applications. The possibility of applying a model checker to RTSJ has great appeal given the complexity and safety requirements of its intended applications. 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 potential future extensions. The implementation relies on a discrete event simulation library, which enables modeling and verification of an RTSJ application under a programmed test environment. The primary advantage of this approach is the possibility of direct execution of the combined model on ordinary Java systems (without the benefit of state backtracking or cost accounting); the primary drawback is the difficulty of implementing important RTSJ features such as non-heap memory areas and asynchronous control transfers. The utility of a general model checker such as JPF in finding RTSJ logic and timing errors is discussed, as well as opportunities presented by JPF for more advanced forms of program analysis such as symbolic execution and test input generation.

Categories and Subject Descriptors

I.6.3 [**Simulation and Modeling**]: Applications; D.4.1 [**Process Management**]: Scheduling; D.3.2 [**Language Classifications**]: Object-oriented Languages

General Terms

Discrete event simulation, real time systems, software verification

Keywords

Real time Java, model checking

1. OVERVIEW

The possibility of using Real Time Specification for Java (RTSJ) [9] 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 compile time static structure, hence 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. In particular, many of the dynamic features of RTSJ in their full generality are beyond the scope of current worst-case execution time (WCET) analysis techniques. While RTSJ programmers can in principle restrict themselves to an RTSJ subset amendable to WCET analysis, this would significantly reduce the appeal and advantages of using RTSJ over existing real time languages. We report here on an application of the Java PathFinder model checker (JPF) [23, 13] 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 event simulation (DES) 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 DES programming (a summary of DES concepts

is given in §5). 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, e.g., order of events scheduled at identical times. Point (a) permits closing an important causality loop impossible on an ordinary JVM:

thread execution cost → deadline misses → miss events → event handlers → 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. MODEL CHECKING AS A VERIFICATION TECHNIQUE

Verification of a software or hardware system can be approached in a variety of ways, ranging from extensive testing (the most widely used technique for software in practice) to formal proofs of correctness. Typically correctness properties are divided into *safety* (nothing bad happens) and *liveness* (something good eventually happens). Formal proofs of correctness are difficult and costly to obtain, especially for non-trivial systems with complex environmental interactions, such as embedded systems. Testing is of course helpful in finding bugs, but cannot provide conclusive evidence of correctness unless a test suite is proven to be comprehensive.

Model checking involves examining all possible states that can arise in an execution of the system under test on given inputs. In many cases, an abstract representation of system states (a *model*) is employed, to reduce both the size and number of states explored – number because abstraction increases the possibility that a new state will be judged to be the same or equivalent to a previously seen state, enabling search pruning.

3. JAVA PATHFINDER AS A JAVA MODEL CHECKER

Although state abstraction can be a powerful tool, it presents several practical challenges, including arriving at an abstraction function that ideally, or at least acceptably, hides unimportant state details while retaining state attributes pertinent to the verification of important correctness conditions. In addition, a transition function must be defined on abstract states, modeling system behavior at the chosen level of abstraction. When the subject is a software system, this transition function is essentially an abstract interpreter. In many cases deterministic transitions on concrete states become nondeterministic transitions on abstract states, diminishing state precision as execution proceeds.

These challenges, plus the availability of increasingly powerful computer resources, have led many researchers to develop explicit state model checkers, in which no state abstraction is employed. Instead of abstract interpreters, these are true interpreters, augmented to provide flexibility and efficiency in state exploration (e.g., backtracking or heuristic search), as well as support for correctness property checking. While safety is typically the primary focus, some model checkers can also deal with liveness properties, e.g., by checking assertions expressed in linear time logic (LTL) [11].

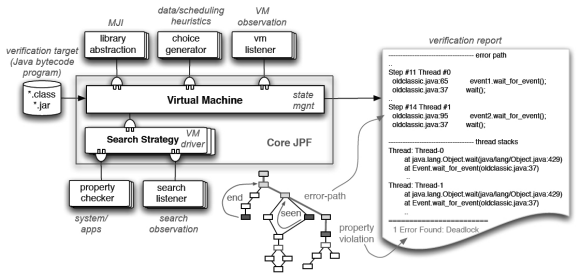

Figure 1: JPF architecture.

Java PathFinder is an explicit state model checker for Java bytecode. JPF focuses on finding bugs in Java programs, such as concurrency errors (e.g., deadlocks or missed signals), Java runtime exceptions such as null pointer dereferences, type (cast) errors, or array out-of-bounds errors. In addition JPF can monitor application specific correctness conditions, in the form of Java assertions or more complex application specific properties. The general architecture of JPF is illustrated in Fig. 1.

4. RTSJ UNDER JPF: REQUIREMENTS AND OBJECTIVES

When one considers applying JPF to RTSJ, 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, constructing and verifying 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 DES 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 DES.

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, notably nondeterministic choice points in the embedding code providing greater test coverage.

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

The fundamental concepts of DES (as developed in the Simula system of the 1970's [4]) 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.
- DES programming simulates concurrency on a sequential computer. One consequence of this strategy is an ironic relationship between real time and simulated time:
  - All real time is expended while simulation clock is unchanging, and
  - Simulation time advances with no real time cost – by discrete increments to the simulation clock.

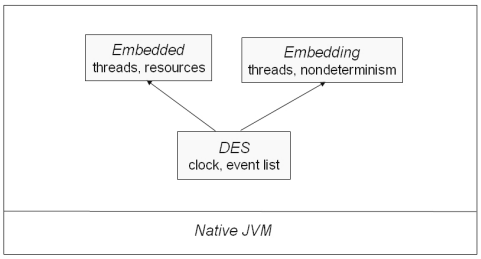

Figure 2: RTSJ architecture for native Java.

```
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();
  }
}
```

Figure 3: The implementation of hold(t).

Since RealtimeThread's are constrained to run as coroutines, the JVM scheduler has only one scheduling choice possible, and DES event based scheduling is used in an *outboard* manner to orchestrate thread interleaving. Fig. 2 summarizes the architecture when running under native Java. 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. For example, Fig. 3 gives our implementation of hold(RelativeTime t), which suspends the execution of a real time thread for a specified time period.

As mentioned in §4, all RealtimeThread interactions are achieved by contention for Resource objects, e.g., a *CPU*. In particular, if only priority inheritance resources are used, the dynamic priority of a RealtimeThread is equal to the maximum of its base priority and the priorities of the threads waiting for priority inheritance resources it possesses (more on this in §7). Synchronized methods have their bodies translated to synchronized statements, and each object that is the subject of a synchronized statement has a shadow Resource object. Hence

```
synchronized ( obj ) { ... }
```

is translated to

```
Robj.seize(); ... Robj.release();
```

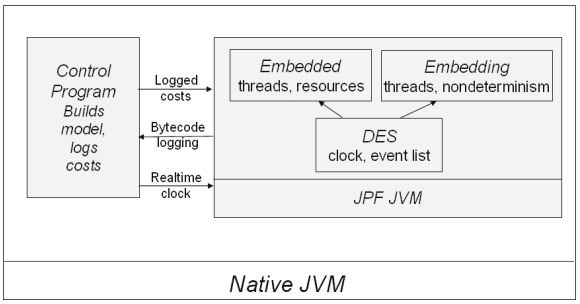

Figure 4: RTSJ architecture under JPF.

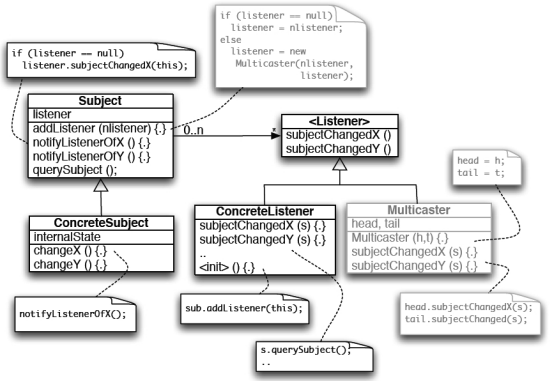

Figure 5: Listener architecture.

where Robj is the shadow Resource for obj. 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. (more on this in §7).

6. STEP 2: COMBINING RTSJ AND JPF

Embedded code written in our RTSJ profile, together with its embedding test code using DES 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, benefiting from the following additional features:

- *Nondeterministic state exploration*, including all orderings of equal priority 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 begins by exploiting two customization features already available in JPF: its JVM *listener interface* [14], and its *Model Java Interface* (MJI) [15] (both features are utilized in the *Control Program* box in Fig. 4).

6.1 JVM Listener Interface

Logging run time (albeit idealized) for Java code under JPF can be done using JPF's JVM listener interface, which

```
public static void main( String args[] ) {
    ...
    JPF.addVMListener(theTestClient);
    ...
}

public void instructionExecuted (VM vm) {
    JVM jvm = (JVM)vm;
    Instruction instruction =
        jvm.getLastInstruction();
    int byteCode = instruction.getByteCode();

    instructionCount++;
    byteCodeCounts[byteCode]++;
    totalCost += byteCodeCosts[byteCode];
}
```

Figure 6: Sample listener code to do cost accounting.

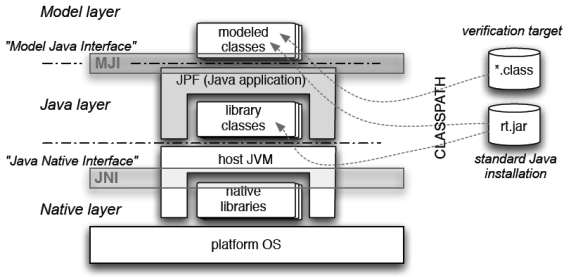

Figure 7: MJI architecture.

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. Note that this cost includes execution times of byte codes executed by intervening methods, e.g., while the handler is suspended waiting for a resource, thus increasing the realism of the overall execution time measure. Fig. 6 shows the listener code used in the JPF RTSJ control program for logging bytecode execution costs. Both the byte code costs are logged for both for the current thread and for all threads.

6.2 Model Java Interface (MJI)

JPF's 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

```
public class JPF_gov_nasa_jpf_rtsj_TestClient{

    public static int getCost(
                  MJIEnv env, int objRef ) {
        return
            gov.nasa.jpf.TestClient.getCost();
    }

    /*
     * so model can have access to
     * true real time clock
     */
    public static long millisSinceEpoch(
                  MJIEnv env, int objRef ) {
        GregorianCalendar gc =
            new GregorianCalendar();
        return gc.getTimeInMillis();
    }
}
```

Figure 8: MJI proxy class to execute code on native JVM.

method. This can trigger the invocation of an *overrun event handler*, which must execute within the JPF JVM.

Fig. 7 depicts JPF's overall MJI architecture, while Fig. 8 illustrates two methods in the JPF RTSJ control program for delivering the total observed bytecode cost and for accessing the native JVM's current real time clock (new GregorianCalendar() in the JPF JVM does not provide access to a real time clock, since that would be meaningless under backtracking).

In the future we hope to address 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 §12.

6.3 Simulated Time vs. Logged Execution Time

An example of an issue that straddles this boundary is the relationship between simulated time and a thread's execution time logged as described above. Here there is a range of realism vs. overhead choices.

- The most accurate (and most expensive) is to execute a hold(t) for each byte code with cost t. However, this would require in effect a conversion of the JPF JVM to execute as a RealtimeThread, so that it would be eligible for scheduled event control under the DES package.

- The least accurate (and least expensive) yet still useful approach is to execute handler byte codes at zero simulated time cost (i.e., in bursts), accumulating the logged cost, and then do a *lumped parameter* hold(t) for the total logged time t prior to a thread's termination.

- We have implemented an intermediate choice on this spectrum whereby hold(t) operations are invoked at all points where threads can interact e.g., before resource seize or release operations, where t is obtained dynamically by logging the JPF byte code execution time since the last such hold(t). To use a financial trading metaphor, the time bookkeeping local to a thread is *marked to market*, i.e., made globally consistent, just prior to any potential interactions with other threads. In the case of a release operation, for example, preceding it by an appropriate hold(t) operation would cause the thread's accumulated byte code execution cost to be accurately exerted in consumed simulated execution time.

7. 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.

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

2) *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.

3) *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).

4) *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).

5) *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 DES programming. 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. Skeletal code

```
public static void holdWithResource(
    RelativeTime holdTime, Resource resource)
      throws InterruptedException {
  RealtimeThread thread =
    (RealtimeThread)Thread.currentThread();
  AbsoluteTime originalStartTime =
    DES.currentTime();
  RelativeTime holdCompletedSoFar =
    new RelativeTime( 0, 0 );

  // loop because may take several periods to
  // complete holdTime if resource is stolen by
  // higher priority thread
  while ( holdCompletedSoFar.compareTo(
      holdTime ) < 0 ) {
    resource.seizeIfNotHeld();
    AbsoluteTime cycleStartTime =
      DES.currentTime();
    DES.hold( holdTime, true );

    RelativeTime timeElapsedThisHold =
      DES.currentTime().subtract(
        cycleStartTime );

    // apply period resource was possessed
    // to holdTime
    holdCompletedSoFar.add( timeElapsedThisHold,
      holdCompletedSoFar );
  }
  // may or may not possess resource at
  // this point
  resource.releaseIfHeld();
}
```

Figure 9: Hold operations on preemptive resources

for this variation of hold is given in Fig. 9. Modeling a CPU resource (say, c) managed under any of these policies is simply done – the code of each RealtimeThread is bracketed by c.seize() and c.release() operations, and all hold(t) operations are replaced by holdWithResource(t, c) operations. All five policy implementations easily generalize to multiprocessing systems by managing pools of CPU resources.

8. APPLICATIONS

Two sample applications of our RTSJ implementation are presented in [18].

8.1 A Multiprogramming Operating System

The first is a simple model of a multiprogramming operating system (OS), where jobs represented by RealtimeThread's contend for a CPU, which is a resource of one of the five types discussed in §7. 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). A fixed job mix was analyzed using our RTSJ implementation in JFP, using CPU's of each of our five resource types. In a sample 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 obtained:

- 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 also given).
- 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.

8.2 Intersection Crossing

The example in §8.1 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 integer priorities chosen from 1 to 8. The intersection is represented by four sectors, each an independent resource.

The intersection transit rules are complex but deadlock free, which as been confirmed (for specific scenarios) by exhaustive search using 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 rapidly approaching, and speeds up accordingly. The *preemption* case is physically impossible!

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

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

Figure 10: Run times for the multiprogramming example under JPF nondeterministic search (backtracking over 24 paths).

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* one car terminates without handler invocation, one car terminates with cost overrun handler invocation, and one car terminates with both handlers invoked.

9. USAGE MODES AND PERFORMANCE

We now present sample 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 §8. Run time figures for the multiprocessing operating system example in §8.1 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).

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 §8.1 Results are shown in Fig. 10.

10. CRITIQUE OF APPROACH

Before concluding with consideration of future and related work, we summarize the principal advantages and disadvantages of our simulation based approach.

The advantages include: (i) modeling embedded and embedding code in a combined framework; (ii) an ability to execute RTSJ programs on ordinary Java systems during development with significant speed up, but loss of nondeterministic search and execution time instrumentation; (iii) use of nondeterminism, both explicit for increased test case coverage, and implicit to examine all scheduling orders; (iv) availability of JPF's property checking facilities to express and monitor application specific correctness conditions, and (v) the generality of the JPF framework for performing more advanced analyses, discussed in §13.

The disadvantages are clearly: (i) the execution slowdown inherent in JPF's JVM-within-a-JVM architecture; (ii) the difficulty of implementing important RTSJ features such as non-heap memory areas and asynchronous control transfers (discussed in §12); (iii) the need to adhere to an idiomatic programming style, e.g., use of explicit CPU objects, and (iv) the limitations of a simulation environment, e.g., no simple migration path to more realistic test environments using true hardware sensors and actuators operating in real time.

11. SCALABILITY

Any model checker, especially one performing full execution on explicit states, is vulnerable to memory exhaustion due to state space explosion. This comes in two forms: linear overhead on representation of individual states in support of execution path exploration (e.g., backtracking), and more significantly the exponential cost of retaining states so that subsequently explored paths resulting in the same state can be detected and cut off.

While our RTSJ implementation in JPF is not exempt from these burdens, several possibilities exist to ameliorate its impact. The most obvious is to retain abstractions of previously encountered states rather than exact states. This not only economizes on memory required for each state, but also increases the likelihood that subsequently encountered states will be judged to have been previously "seen", thereby pruning the execution path search. In particular, we conjecture that the extremely fine grain representation of time in RTSJ (to nanosecond precision) makes the probability of exact state reoccurrence on alternative paths vanishingly small. Ideas for state abstraction are sketched in §13.2. Other state explosion countermeasures are symbolic execution (essentially abstraction with adaptive refinement), and heuristic search (ranking a bounded number of paths to be pursued by estimated merit); these are also surveyed in §13.2.

It is an open research question whether space economization techniques for features specific to RTSJ can be developed without compromising vigilence on key application safety and timing properties. We hypothesize that conservative design practices in well-engineered embedded systems may imply that appropriate state abstractions can ensure that representatives of the most critical system states will not be overlooked.

Finally, we observe that our approach to verifying both embedded and embedding code in a combined manner is both a strength and a weakness – a *strength* because both the system under test and its operational environment are verified in a comprehensive manner, and a *weakness* because nondeterminism used in the embedding code to increase test case coverage can aggravate state space explosion. This weakness implies that any techniques for "steering" nondeterministic choices within the embedding code to cases most "stressful" to the embedded code could be very helpful. We have preliminary ideas in this regard, taking a *game playing* viewpoint on nondeterministic choices [21]. Under this metaphor, the embedding code is an adversary of the embedded code, seeking to choose stimuli that drive the embedded code to cost and deadline overruns. For its part, the embedded code could attempt to counter this "attack" by

adaptive techniques such as slack time based scheduling.

12. FEATURES NOT EASILY IMPLEMENTED UNDER THIS APPROACH

Our approach to implementing RTSJ without JPF JVM modifications exploited two crucial architectural features as mentioned in §6.1: (i) the JPF JVM listener interface, and (ii) the Model Java Interface (MJI). The result is a surprisingly large subset of RTSJ features can be supported under this approach. (albeit with some coding idioms, see e.g., §5).

In §6.2 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. One potential avenue for the second is to apply byte code transformation to inject interrupt exceptions [22].

13. FUTURE POTENTIAL OF USING JPF

Now examine recent JPF developments and their potential for augmenting analysis of RTSJ programs.

This application breaks new ground for JPF 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.

13.1 Programmable Thread Schedulers

Fig. 11 illustrates the two major concepts in the class hierarchy of the JPF JVM: Search and VM. The first is the driver of the second, and also maintains vigilence over correctness properties. The VM is the state generator, operating under the control of the Search. Fig. 12 shows how the Search class hierarchy can be used to implement particular search strategies, e.g., depth-first and heuristic, with an associated sorted state queue.

At present there are two types of execution path variation in JPF: thread scheduling choices (which ready thread to run next; which thread to wake up in notify()), and explicit nondeterminism, e.g., gov.nasa.jpf.jvm.Verify.randomBool(), which generates two alternative execution paths, one returning true and one returning false.

Plans are underway to unify nondeterministic search and scheduling via programmable *choice generators* (see Fig. 13). This will enable user controllable interactions between thread scheduling selections and search management. It is believed that this new form of JPF extensibility will permit the resource-centric "outboard" scheduling mechanisms described in §7 to be moved into JPF's core thread scheduler, where they properly belong. A major benefit would be a greater ability to run RTSJ applications without translation to our resource contention based programming idiom. In particular, the explicit CPU objects described in that section would no longer be necessary.

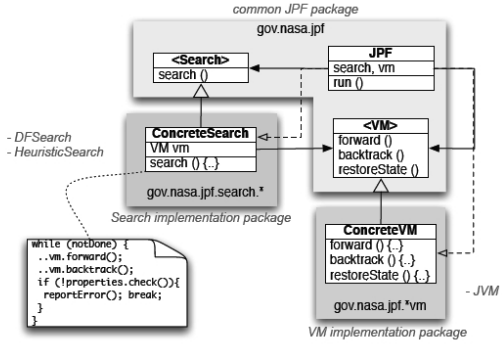

Figure 11: Search and VM as cooperating JPF classes.

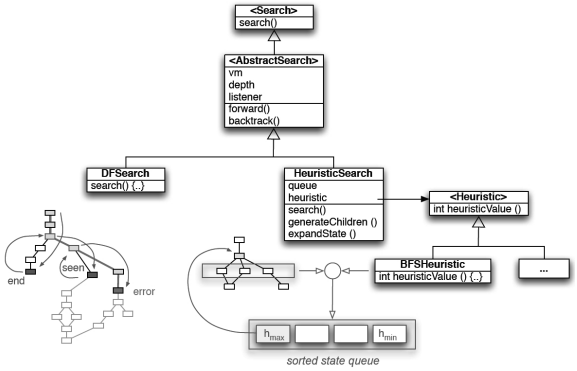

Figure 12: Implementing particular search strategies.

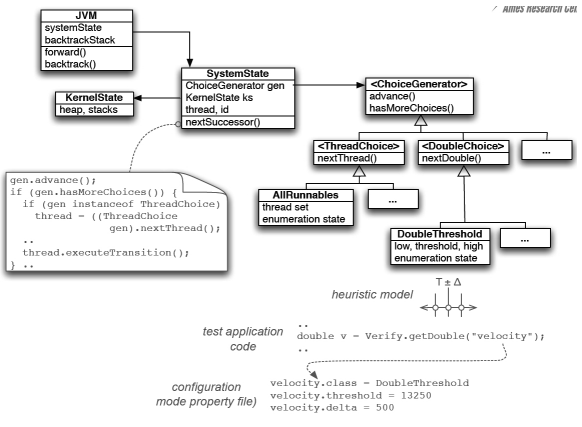

Figure 13: Choice generators as a means of implementing RTSJ scheduling.

13.2 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 [10]. 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 [16]. 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 [24]. 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 [7]. Conditions revealing such flaws in real time checklist procedures might be determined by symbolic execution in this manner.

14. RELATED WORK

Model checking of timed automata representations has become very popular ([2]; see [3] for a good overview) for the analysis of real time systems. Our approach differs in that we are analyzing systems with complex transitions but simple explicit timing information, whereas in the timed automata approach is typically applied to analyze systems with complex timing, but simple transitions (e.g., between abstract states in given time intervals). By contrast we are performing genuine program execution (not abstracted, or symbolic). The notion of applying timed automata style reasoning is appealing, but represents a major new line of research, due to the complex transitions in our program executions, e.g. memory allocation, exception handling, etc.). Our emphasis at present is checking program safety properties including scheduling errors such as priority inversion, as well as classic Java errors such as uncaught exceptions and assertion violations.

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 [19]. Tools are available to benchmark RTSJ implementations [6].

Model checking is a vigorously evolving research area. Bandera [1], Bogor [8], and the work of Bart Jacobs et al. on *JavaCard* verification [12] are examples of model checking applied to Java programs. A closely related area is run time verification of Java systems [17]. 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) [11]. RTSJ itself is drawing critical and insightful analysis, such as the work on Ravenscar [5, 25].

Finally, the advent of the Java Platform Debugger Architecture (JPDA) offers the potential of greatly improved flexibility and performance for our dual JVM implementation strategy. The idea of using a debugging interface for model checking has been examined for the Gnu debugger gdb by Mercer and Jones in [20]. Major research issues are presented by implementing state saving and backtracking under this approach. Moreover, the challenges of implementing the RTSJ features missing in our system, e.g., memory varieties and ATC, would still be present — unless an RTSJ compliant JVM could be obtained that supports JDPA, which seems unlikely.

Acknowledgements

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

15. REFERENCES

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

[2] G. Behrmann, K. G. Larsen, and J. I. Rasmussen. Optimal scheduling using priced timed automata. *ACM SIGMETRICS Performance Evaluation Review*, 32(4):34–40, March 2005.

[3] J. Bengtsson and W. 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.

[4] G. M. Birtwistle, O.-J. Dahl, B. Myhrhaug, and K. Nygaard. *Simula BEGIN*.

Auerbach/Studentliteratur, Philadelphia, 1973.

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

[6] A. Corsaro and D. 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.

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

[8] M. B. Dwyer, J. Hatcliff, M. 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.

[9] T. R.-T. for Java^{TM} Expert Group. https://rtsj.dev.java.net.

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

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

[12] 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.

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

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

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

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

[17] M. Kim, S. Kannan, I. Lee, and O. Sokolsky. Java-MaC: 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.

[18] G. Lindstrom, P. C. Mehlitz, and W. Visser. Model checking Real Time Java using JavaPathfinder. In *Proc. Third International Symposium on Automated Technology for Verification and Analysis (ATVA)*, pages 444–456. Springer Lecture Notes in Computer Science, Oct. 4–7 2005. vol. 3707.

[19] C. D. Locke. Real-Time Java moving into the mainstream. *RTC Journal*, January 2004.

[20] E. Mercer and M. Jones. Model checking machine code with the GNU debugger. In *Proc. SPIN Workshop*, 2005.

[21] S. Shoham and O. Grumberg. Multi-valued model checking games. In *Proc. Third International Symposium on Automated Technology for Verification and Analysis (ATVA)*, pages 354–369. Springer Lecture Notes in Computer Science, Oct. 4–7 2005. vol. 3707.

[22] W. Tao. *A Portable Mechanism for Thread Persistence and Migration*. PhD thesis, Univ. of Utah, 2001.

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

[24] W. Visser, C. S. Păsăreanu, and S. Khurshid. Test input generation with Java PathFinder. In *Proceedings of ISSTA*, July 2004.

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

APPENDIX

A. JPF RTSJ PROFILE

From RTSJ 1.0.1(b); * denotes not implemented

1. *Threads:* NoHeapRealtimeThread*, RealtimeThread.
2. *Scheduling:* AperiodicParameters, ImportanceParameters, PeriodicParameters, PriorityParameters, ProcessingGroupParameters, ReleaseParameters Schedulable, Scheduler, SchedulingParameters SporadicParameters.
3. *Memory Management:* GarbageCollector*, HeapMemory, ImmortalMemory, ImmortalPhysicalMemory, MemoryArea, MemoryParameters, PhysicalMemoryManager*, PhysicalMemoryTypeFilter*, RawMemoryAccess, RawMemoryFloatAccess, ScopedMemory*, SizeEstimator*, VTMemory*, VTPhysicalMemory*.
4. *Synchronization:* MonitorControl*, PriorityCeilingEmulation*, PriorityInheritance*, WaitFreeDequeue* – deprecated, WaitFreeReadQueue*, WaitFreeWriteQueue*.
5. *Time:* AbsoluteTime, HighResolutionTime, RationalTime* – deprecated, RelativeTime.
6. *Clocks and Timers:* Clock, OneShotTimer, PeriodicTimer, Timer.
7. *Asynchrony:* AsyncEvent, AsyncEventHandler, AsynchronouslyInterruptedException*, BoundAsyncEventHandler, Interruptible*, Timed.