

Model Checking Real-time Java

Pavel Parizek¹, Tomas Kalibera¹, and Jan Vitek²

¹ Department of Software Engineering, Charles University

² Computer Science Department, Purdue University

Abstract. The main challenges in model checking real-time Java programs are scalability and compliance with the Real-time Specification for Java (RTSJ) semantics. A model checker for RTSJ programs has to support the notion of thread periods and priority-preemptive scheduling as well as understand the region-based memory model introduced by the specification. We present the R_J model checker for multi-threaded RTSJ programs, which is based on Java PathFinder. R_J explores execution paths that correspond to priority-preemptive schedules and involve valid sequences of thread releases with respect to their periods and priorities. The other novel features of our work are an abstraction of time based on thread periods, and a model of RTSJ memory regions and API. We evaluated our approach on real-time benchmark that models an aircraft collision detection algorithm implemented in real-time Java. Thanks to our precise account of RTSJ semantics, R_J was able to verify the application and reported no spurious errors.

1 Introduction

While most of today’s safety-critical embedded systems are written in C, the recent entry of Java-based solutions in this market along with a gradual move by regulatory bodies towards formal methods is creating opportunities to reshape how safety-critical systems are engineered. Java is inherently safer than C as it prevents many memory-related errors (dangling pointers, unsafe casts) and mandates run-time checks for other (array bounds, stack overflow). Moreover, unlike C, concurrency is part of Java and is rigorously specified in terms of a memory model. The behavior of concurrent programs can thus be understood without any knowledge of the compiler, operating system, or hardware architecture. In terms of performance on real-time benchmarks, Java compilers generate code that is competitive with C, making Java suitable as a development language for real-time systems and a target for autocoders. For the remainder of this paper, we assume a uniprocessor setting, as this is still the platform of choice for safety-critical systems.

The goal of our research is to explore techniques for automatic verification of real-time programs written in Java. In particular, we focus on extensions included in the Real-time Specification for Java (RTSJ) [1], as they provide a standardized API for real-time computing on the Java platform. The challenges that have to be solved pertain to dealing with multi-threading, real-time scheduling, thread

periods, and region-based memory management. In the RTSJ, programs can dynamically allocate new objects and create new threads at any time, and they often do so in practice. Furthermore, any solution must be scalable as real systems are often quite large. Testing real-time applications is particularly difficult because they are often developed on general purpose computers and deployed on specialized embedded devices. The timing characteristics of the development machine are often completely different from that of the target platform. Thus, some concurrency errors may only manifest on the target platform. Providing tools that let developers find errors before deployment is of great practical interest.

State of the art model checkers are of little help. While some checkers target real-time, they verify high-level abstract models of real-time programs rather than their implementation [8, 11]. General purpose Java model checkers (e.g. [3]) capture key aspects of Java program behavior, including dynamic allocation and threads, but do not understand the semantics of the real-time extensions of Java. In particular they have no understanding of real-time scheduling, priority inversion avoidance protocols, and thread periods, and they do not support the region-based memory management. This will lead them to explore execution paths involving infeasible thread interleavings and/or fail to discover some errors.

This paper describes R_J , an explicit state model checker for real-time Java programs that aids developers in performing exhaustive, platform independent, testing of concurrent real-time applications. We have chosen to base R_J on an existing model checker, Java PathFinder (JPF) [6]. JPF is an explicit state model checker for Java. As it does not have built-in support for the real-time Java APIs, our first contribution was to extend it to support region-based memory management, priority inheritance locks, and important parts of RTSJ API. These are required to understand the semantics of any real-time Java program. Our second contribution is an algorithm for priority-preemptive scheduling of real-time threads in the context of state space traversal. The algorithm covers all feasible thread interleavings and avoids interleavings that are not feasible given specific thread priorities and periods, under the assumption that deadlines are met. As JPF operates on Java bytecode, in a platform independent way, R_J cannot provide precise clock values to the program being checked. A realistic timing model would require a detailed model of the hardware architecture, compiler optimizations and operating system. We have thus defined an abstraction of time that only depends on scheduling of periodic threads — this is our third contribution.

Lacking a precise timing model, R_J does not support safety properties that depend on exact clock values, it is limited to *time-independent safety properties*. Specifically, R_J can detect invalid usage of RTSJ APIs, invalid usage of memory regions (e.g. invalid assignments), and general runtime errors such as assertion violations and race conditions. We evaluate the scalability of our approach on CD_x , an implementation of a real-time aircraft collision detection algorithm in Java. We emphasize that it is not possible to verify this program using the original JPF tool, since JPF does not understand RTSJ semantics and thread periods.

2 Background

Program state space in JPF. The state space of a Java program in JPF is a tuple $\langle S, \alpha, s_0, F \rangle$, where $S = \{s_0, \dots, s_n\}$ is a set of states, $\alpha \subseteq S \times TR \times S$ is a transition relation, s_0 is an initial state, F is the set of final states, and TR is the set of transitions. A path p in the state space is a sequence $s_0, tr_1, s_1, \dots, tr_n, s_n$, where s_i are states and tr_i are transitions. Each transition corresponds to a sequence of bytecode instructions executed by the same thread. We use the lower index i in tr_i only when the order of transitions in a state space path is important. Similarly, we use the symbol tr^k when it has to be explicitly denoted that a transition is executed by the thread T_k . A transition tr , such that $(s, tr, s') \in TR$, is terminated when an instruction is executed that (a) reads or modifies the global state visible to all threads (e.g. accesses a shared variable on the heap), (b) changes the status of any thread, (c) performs a synchronization operation (e.g. attempts to acquire a lock), or (d) performs non-deterministic data choice. In the cases (a)-(c), we say that the instruction represents a *scheduling-relevant operation* and triggers a *thread choice point* that is associated with the state s' . Similarly, in the case (d) we say that the state s' is associated with a data choice point. For each state $s \in S$ associated with a thread choice point, the *set of enabled transitions* contains a transition for each thread that can be scheduled in the state s (e.g. for each runnable thread). Formally, the set is defined as $\{(s, tr^{k_1}, s_1), \dots, (s, tr^{k_n}, s_n)\}$, where k_1, \dots, k_n are indexes of threads that can be scheduled in s .

Real-time Java. The RTSJ extends the Java platform with new libraries, it extends the behavior of the virtual machine and redefines semantics of some existing language features. For our purposes, the key features of the RTSJ are limited to the new threading model, priority-preemptive scheduling, and a region-based memory model that sidesteps garbage collection. The threading model introduces *real-time threads* which are strictly scheduled according to their priorities — priority-preemptive scheduling policy is used by default. When a running thread is preempted, the runnable thread with the highest priority is selected. A running thread is only preempted if: (i) it explicitly yields the processor, (ii) it blocks on a locked monitor, or (iii) a thread with a higher priority becomes runnable. Threads with equal priority are scheduled in the FIFO order — there is a queue for each priority, and also a queue of sleeping and blocked threads. When a sleeping thread becomes runnable, it is added to the tail of the queue associated with its priority. Although Java has a notion of thread priority, it does not require schedulers to pay attention to priorities or to restrict preemption — this is one of the key additions of the RTSJ. This implies that if the highest priority thread does not yield the processor, no low priority thread will ever be scheduled. An important characteristic of a real-time thread is its deadline. A real-time thread is required to complete its execution before the deadline, otherwise a deadline miss occurs. Threads can be periodic or aperiodic. A periodic thread has a fixed *period*, and at any point in its execution it may explicitly choose to sleep (i.e. yield the processor) for the rest of its period by calling `wait-`

ForNextPeriod. We say that a periodic thread is *released* (becomes runnable and is added to the tail of the queue associated with its priority) at every start of its period. The period of a thread is typically also its deadline. The first release of a periodic thread can be set to happen at a particular absolute time, or it can be specified relatively based on the current time.

We use the following notation: P_i denotes the period of a thread T_i , R_i denotes its first release time, and Pr_i represents its priority.

3 Motivating Examples

In this section we illustrate the challenges of verifying real-time Java programs. Figure 1 is an instance of the producer-consumer design pattern. The `main` method starts two periodic subclasses of `RealtimeThread` which communicate through a shared buffer. The `Producer` allocates and adds to the buffer a new message string every 10 milliseconds. The `Consumer` checks the buffer every 5

```

class Example {
    static class Consumer extends RealtimeThread {
        public void run() {
            while (waitForNextPeriod()) {
                synchronized (buffer) {
                    if (buffer.isEmpty()) continue; else buffer.remove(0);
                }
                if (--left <= 0) break;
            }
        }
    }
    static class Producer extends RealtimeThread {
        public void run() {
            for (int i = 0; i < 100; i++) {
                synchronized (buffer) { buffer.add(new String(i)); }
                if (!waitForNextPeriod()) break;
            }
        }
    }
    static final List buffer = new ArrayList();
    static int left = 100;

    public static void main(String[] args) {
        new Consumer(new PriorityParameters(19),
            new PeriodicParameters(null, new RelativeTime(5, 0))).start();
        new Producer(new PriorityParameters(15),
            new PeriodicParameters(null, new RelativeTime(10, 0))).start();
    }
}

```

Fig. 1. Motivating example: producer-consumer in RTSJ

milliseconds. The program terminates when 100 messages have been consumed. The `Consumer` has a higher priority, 19, than the `Producer`, 15.

The challenge for model checkers such as JPF is to avoid exploring paths that are prohibited by the semantics of RTSJ. For instance, with the wrong scheduling policy, the model checker could explore an execution path in which a lower-priority thread is run when higher priority thread is runnable. In the example, there are only two possible points of preemption: the call to `waitForNextPeriod` which lets another thread run, and the `synchronized` statement. By default, RTSJ monitors implement the priority inheritance protocol. Thus if the producer holds the lock when the consumer tries to acquire it, the producer's priority will be raised long enough to let it complete its critical section.

Sound treatment of RTSJ semantics and scheduling is, however, not possible without understanding of thread periods. JPF completely ignores time and abstracts timed sleeps. When a thread calls, e.g., `Thread.sleep`, the current transition is terminated, a thread choice point is created, and the thread remains runnable as if the time for sleep had already passed. The only consequence of a sleep call is that other threads can be scheduled. In the example, once JPF implemented the RTSJ scheduling policy correctly, JPF would explore a single execution path, in which the high-priority `Consumer` loops forever. This is a consequence of the fact that the `Consumer` thread is selected at each choice point due to its priority. Although the `Producer` thread is runnable at some states, it will never be selected due to the thread's lower priority. Note that although the single explored execution path is infinite, JPF will actually explore only one iteration of the loop in `Consumer`'s body and then terminate the state space traversal. This is because a transition is terminated at the call of `waitForNextPeriod`, and the successive states of the example program at two successive calls of `waitForNextPeriod` in the run method of `Consumer` are equivalent. All transitions involving the `Consumer` thread are explored in such a case, and there are no unexplored transitions for the `Producer` thread in the state space — moreover, no transitions associated with the `Producer` thread are enabled in any state. To summarize, we argue that it is necessary to support the priority-preemptive scheduling and also the notion of thread periods. Otherwise, a model checker would (a) explore execution paths involving infeasible thread interleavings and report spurious errors, or (b) fail to explore some feasible behaviors with respect to the RTSJ semantics.

Another aspect of RTSJ that has to be taken into account is its special memory model. To avoid the timing hazards introduced by garbage collection, real-time Java supports a form of region-based memory allocation. Objects allocated in a `MemoryArea` are reclaimed all at once when the memory area is not in use. While it is always safe for objects within the same region to reference each other, certain cross-region references can lead to dangling pointers. In order to prevent this, the virtual machine throws a run-time exception when a program attempts to create an unsafe reference. Practical experience with the `MemoryArea` API has shown that it is error-prone and thus its usage must be checked. Figure 2 shows a program involving invalid assignment of references between

```

class AssignmentExample {
    static class LogicWithError implements Runnable {
        public void run() {
            Integer zeroInScope = new Integer(0);
            bufferInImmortal.add(zeroInScope);
        }
    }
    static List bufferInImmortal = new ArrayList();

    public static void main(String[] args) {
        LTMemory scope = new LTMemory(1024, 1024);
        new RealtimeThread(scope, new LogicWithError()).start();
    }
}

```

Fig. 2. Motivating example: invalid assignment of references

memory areas. A reference to an object `zeroInScope` allocated in a short-lived scoped memory area `scope` is stored in the `bufferInImmortal` variable that is allocated in the immortal memory area. When the memory area `scope` is reclaimed, the reference to `zeroInScope` stored in `bufferInImmortal` will become dangling.

4 Model Checking RTSJ Programs

This section describes our approach to model checking multi-threaded real-time Java programs against time-independent safety properties. The input to \mathcal{R}_J is a real-time Java program that consists of one or more periodic threads and a main thread, which performs some initialization, starts all periodic threads, and then waits for the periodic threads to terminate. We make the following assumptions: (A1) All periodic real-time threads meet their deadlines; (A2) The program runs on a single processor and therefore the interleaving semantics of concurrency can be used; (A3) First release times of all periodic threads are specified relatively in the program code. Assumption A1 is realistic for a majority of real-time programs, as the lack of deadline misses is being extensively verified both by testing and analysis, and A2 is typical nowadays. The assumption A3 is realistic, first release times are usually not absolute. If they were, the code could be easily modified to use relative times for the purpose of checking with \mathcal{R}_J .

4.1 Thread Scheduling

In JPF, thread scheduling is a key component of the state space traversal algorithm. At any thread choice point, the set of enabled transitions contains one element for each thread that can run according to the scheduling policy. Non-deterministic choice among the enabled transitions guarantees that all possible schedules are explored. JPF adheres to the concurrency model of Java, which

assumes unlimited number of processors and imposes no restrictions on scheduling and preemption, and therefore each runnable thread can be selected in each state.

The state space traversal algorithm in R_J performs priority-preemptive scheduling and considers thread periods. This means R_J (i) schedules threads according to their priorities and (ii) releases sleeping threads in the correct order with respect to their periods. While priority-preemptive scheduling is straightforward, generation of valid sequences of thread releases with respect to periods is more difficult. Since R_J does not use a timing model of a particular platform, it cannot determine the exact clock value at all points in the program execution and therefore cannot release sleeping threads based solely on the clock. Let $r_i(p)$, $r_i(p) \geq 0$, be the number of releases of the thread T_i on p (since the program start) and $r_{i,j}(p)$, $r_{i,j}(p) \geq 0$, be the number of releases of T_i on p since the last release of T_j . The key steps of the algorithm are shown in Figure 3. For conciseness, we focus only on thread scheduling choices and completely neglect non-determinism related to data. Thorough description of the algorithm follows.

```

visited = stack = path = {}; push(stack, s0); explore(s0)

procedure explore(s) {
  if error(s) counterexample = stack; terminate
  for (tr ∈ enabled(s, path)) {
    s' = execute(tr)
    if not (s' ∈ visited) {
      visited = visited ∪ s'
      push(stack, s'); push(path, tr)
      explore(s')
      pop(path); pop(stack)
    }
  }
}

procedure enabled(s, p) {
  W = {Ti : Ti is sleeping in s} // set of sleeping threads
  L = {Ti : Ti is runnable in s} // set of runnable threads
  PrL = maxTi ∈ L Pri // maximum priority of runnable threads
  E = {trr : Tr ∈ L . Prr ≥ PrL}
  for (w ⊆ W) {
    save_release_counts(p)
    for (Ti ∈ w) r•,i(p) = 0
    for (Ti ∈ w) ri,•(p) = ri,•(p) + 1
    for (Ti ∈ w) ri(p) = ri(p) + 1
    if ValidReleases(s, p) E = E ∪ {tru : Tu ∈ w . Pru ≥ PrL}
    restore_release_counts(p)
  }
  return E
}

```

Fig. 3. State space traversal algorithm in R_J

Valid sequences of thread release events are determined on the basis of the number of times a thread can be released between any two consecutive releases of any other thread (assuming that no deadlines are missed). For each pair T_j, T_k of threads such that $P_k \geq P_j$, after a certain number of releases of T_j and T_k , T_j can be released exactly $\lfloor P_k/P_j \rfloor$ or $\lfloor P_k/P_j \rfloor + 1$ times between any two consecutive releases of T_k . For example, given two threads T_a and T_b with periods $P_a = 35$ ms and $P_b = 10$ ms, then T_b can be released 3 or 4 times between any two releases of T_a . This constraint on thread releases is expressed in the state space traversal algorithm by the predicate $ValidReleases(s, p)$, which is defined as follows (as of now, we only use the upper bound $\lfloor P_k/P_j \rfloor + 1$ mentioned above, to make the algorithm simpler):

$$ValidReleases(s, p) \equiv \forall T_i, T_j : (P_j \geq P_i) \Rightarrow r_{i,j}(p) \leq \lfloor P_j/P_i \rfloor + 1$$

The predicate does not allow to determine the precise moment at which a thread should be released, nor when a new thread should be started. To answer these questions, we introduce the concept of *non-deterministic release*. In a state s on the path p , the sets L of runnable threads in s and W of sleeping threads in s are created. For each $w \subseteq W$, it is determined whether all threads in w can be safely released (released at the same moment of time) — first, all counters of past releases for all threads in w are updated as if they have actually been released together, and then the $ValidReleases$ predicate checks whether the constraints on release sequences based on the numbers of past releases would hold. In particular, the predicate does not hold if any thread T_i would be released too many or too few times between consecutive releases of any other periodic thread. Note that the size of the set w will be very small in practice. Then the set E of enabled transitions in s is constructed as follows. The set E will contain a transition for each top-priority runnable thread $T_r \in L$ and, for each $w \subseteq W$, one or more transitions corresponding to the top-priority threads T_{u_1}, \dots, T_{u_n} from the set $L \cup w$. Assuming that Pr_L is the maximal priority over all runnable threads in L , the set E will contain transitions corresponding to (i) all runnable threads with the top priority Pr_L and (ii) all sleeping threads in W that have greater or

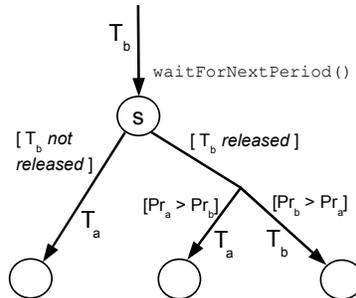


Fig. 4. Effect of non-deterministic release on the shape of state space

equal priority than Pr_L — this way it is expressed in Figure 3. Note that a new periodic thread created in the code of the checked program is initially sleeping and can be released at any time. It is however added to the $r_{\bullet\bullet}(p)$ matrix only after it has been released the first time on the path p . When a transition tr^r corresponding to $T_r \in L$ is selected, then no sleeping thread will be released and T_r will be scheduled. When a transition corresponding to $L \cup w$ for a specific $w \subseteq W$ is selected, then all threads in w are actually released and added to the queues of runnable threads based on their priorities, and the corresponding release counts are updated ($r_{\bullet\bullet}, r_{\bullet}$); one of the top-priority threads from $L \cup w$ will be scheduled. This concept is illustrated in Figure 4 for the special case of two threads T_a, T_b , where T_a is runnable and T_b is put to sleep by call to `waitForNextPeriod`. This way, the priority-preemptive scheduling is honored as well. The state space traversal algorithm then explores all possible choices that correspond to the elements of E — in particular, it is non-deterministically selected whether any sleeping threads are released or not. The algorithm explores all feasible program behaviors that can happen on a particular target platform. It, however, explores also some unfeasible behaviors due to unavailability of the precise timing model (discussed in more detail in Section 4.4) and due to over-approximations in the algorithm.

4.2 Abstraction of Time

RTSJ programs may use clock values in their application logic, and therefore a model of time is needed. Since R_J cannot determine the exact clock value at an arbitrary point in program’s execution due to the unavailability of a platform-independent timing model, we exploit the knowledge of thread periods and numbers of past releases on state space paths to create an abstraction of time. The initial value of abstract clock C is equal to zero, i.e. it is assumed that the main thread starts at the time $C = 0$. When a periodic thread T_i is released for the first time on a state space path p , its first release time R_i is set to the current value of abstract clock at p . When a thread T_i is released for any subsequent time ($r_i(p) > 1$), the abstract clock advances — the current value C of abstract clock is set to $C = R_i + r_i(p) * P_i$. Note that R_i is already defined for a thread T_i when $r_i(p) > 1$, since it was set at the occasion of the first release of T_i . It follows that the first release event on a state space path p happens at the time $C = 0$. Note that abstract clock value defined in this way provides a lower bound on the exact clock value during program’s execution.

4.3 Modeling Semantics of RTSJ API

The state of a Java program, as represented by JPF, consists of the following components: contents of the heap, contents of the static area in which all static variables and class data are allocated, and a set of threads. Each thread has a status and program counter, and it is associated with a stack of method frames. In order to capture the semantics of real-time Java, in R_J we extended the state of a program with the following elements: (a) a FIFO queue of runnable threads

for each possible thread priority; (b) a FIFO queue of sleeping threads; (c) a mapping between objects and their memory area; (d) a stack of scoped memory areas for each thread in the program. Ordered queues are critical to capture the way threads are selected for execution by the scheduler. We also extended the program state with a numeric variable that contains the current value of abstract clock. Nevertheless, the abstract clock value is not considered in state matching – when all components of two states s_1 and s_2 except the clock value are equal, then the states s_1, s_2 are considered as being equivalent by R_J .

More formally, a state of a RTSJ program is a tuple $s = \langle DH, IA, TS, RTQ, STQ, C \rangle$. DH is a sequence $\{do_1, \dots, do_n\}$ of heap allocated objects. $IA = \{io_1, \dots, io_n\}$ is a sequence that represents the content of the immortal memory area — all static variables and class data are allocated in this area. TS is a set $\{T_1, \dots, T_n\}$ of threads. RTQ denotes queues of runnable threads for all possible priorities, STQ denotes a queue of sleeping threads, and C represents the current value of abstract clock. An element $do \in DH$ is a tuple $\langle addr, ma, wtq \rangle$, where $addr$ is a heap address of the object represented by do , ma identifies the memory area in which the object represented by do is allocated, and wtq is a queue of threads blocked on the object’s monitor. A thread is a tuple $T_k = \langle k, SF, st, pc, Pr_k, P_k, R_k, MAS \rangle$, where k is the thread’s index, $SF = \{f_1, \dots, f_n\}$ is a stack of method frames, $st \in \{new, running, blocked, unblocked, interrupted, terminated, sleeping\}$ is the thread status, pc is its program counter, Pr_k is the thread’s priority, P_k is the thread’s period, R_k is the thread’s first release time, and $MAS = \{ma_1, \dots, ma_n\}$ is a stack of memory areas associated with the thread. The initial state s_0 involves the main thread, which has the index 0, default priority 5, and an empty stack of method frames. All other components of the initial state are empty.

Based on the state extensions, we created a model of important RTSJ features, such as the priority inheritance locks and the region-based memory model. In particular, our model of the RTSJ API in R_J keeps track of the key attributes of each thread (period, priority, and first release time), and provides them to the state space traversal algorithm as needed. We also extended the set of operations considered as scheduling-relevant in R_J to include calls of the `waitForNextPeriod` method and calls of methods that change thread attributes. In case of plain Java programs and JPF, an access to a heap object reachable from multiple threads is considered as a scheduling-relevant operation, i.e. a transition is terminated and a thread choice point is created at the corresponding bytecode instruction. While this still applies to objects stored in the heap and immortal memory in R_J , in case of an access to an object stored in a scope memory area R_J also takes into account the visibility of the scope area by multiple threads. A scope memory area is visible by a thread, if it is on the scope stack of the thread. An object is not reachable from multiple threads, when it is stored in a scope memory area that is visible only from a single thread — access to such an object is not a scheduling-relevant operation.

The RTSJ specification introduces several kinds of runtime errors, which correspond mainly to incorrect usage of RTSJ API and memory regions by a

program. Currently, R_J can detect the following RTSJ-specific errors (violations of safety properties) in multi-threaded programs:

- invalid assignments between different memory areas (e.g. attempts to store a reference to an object in a scoped memory area into the heap),
- calls of specific API methods (i) when a target object is in an invalid state or (ii) with invalid arguments, and
- calls of methods specific to real-time threads on plain Java threads.

Note that occurrence of these errors does not depend on exact clock values or precise execution times of program fragments.

4.4 Correctness, Precision, and Termination

Thread scheduling and releases. Usage of the state space traversal algorithm described in Section 4.1 guarantees that, when R_J is used to check a real-time Java program P , all state space paths (thread interleavings) corresponding to feasible executions of P are explored. Implementation of the priority-preemptive scheduling ensures that, for each pair of threads T_l, T_h such that T_l has lower priority than T_h ($Pr_l < Pr_h$), the thread T_l will be scheduled only when T_h is sleeping or blocked. This all follows directly from: (1) the definition of the concept of non-deterministic release, (2) the constraint on thread releasing that is based on periods and numbers of past releases, and (3) the structure of the program’s state space as determined by the set of transitions enabled in each state. As a consequence, all violations of the supported time-independent safety properties can be found by R_J .

When R_J is applied to a program P and terminates without reporting any error, it is guaranteed (i) that all feasible execution paths of P were explored and (ii) that P contains no errors detectable by R_J . However, if a run of R_J fails (e.g., it runs out of memory) or does not terminate, then it may miss some errors. No guarantees on coverage can be given in such a case, since it is not possible to determine precisely how large a part of the state space was explored.

Absence of timing model. Because of the unavailability of a precise timing model and the concept of non-deterministic release, R_J may explore such execution paths of a program P that involve thread interleavings that are not possible on a particular target platform — an over-approximation of the program’s behavior with respect to the timing model of the platform.

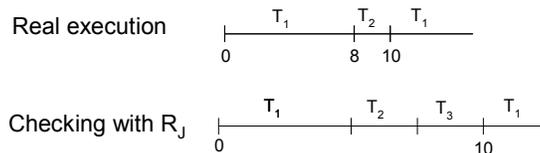


Fig. 5. Deadline misses in real execution and state space traversal

The assumption of no deadlines missed is a key one — it allows us to avoid the pathological example mentioned in Section 3, in which the only execution path explored by the model checker is the one in which the top-priority thread T_p starves out all other threads in the program. Once every periodic thread has to keep its deadline, which is equal to its period, we know that none of the threads can run all the time. R_J will thus sometimes release lower-priority threads instead of higher-priority ones to explore the execution paths where the higher-priority threads wait for their next release. Specifically, the state space in R_J includes also paths in which threads with lower priorities are selected at some thread choice points. Given a state s associated with such a thread choice point, the set of transitions enabled at s contains a transition tr^p associated with the top-priority thread T_p (representing the case when T_p is released) and a transition tr^k associated with some other thread $T_k, Pr_k < Pr_p$ that is runnable in s (representing the case of T_p not being released).

If R_J is applied on a program in which some threads do not meet their deadlines, it would explore infeasible execution paths that correspond to invalid sequences of thread release events. Consider for example the case of a program that contains three threads T_1, T_2, T_3 with the following attributes: periods $P_1 = P_2 = P_3 = 10$, priorities $Pr_1 > Pr_2 > Pr_3$, first release time equal to 0, and execution (computation) times $C_1 = 8, C_2 = 4, C_3 = 4$ in each period. The scenario is illustrated on Figure 5. In a real execution of the program, when all threads are released (e.g., at the time 0), threads T_2 and T_3 will miss their deadlines — this is because the thread T_1 is scheduled first (it has the highest priority) and when it completes its task in a period at the time 8, the other threads cannot complete their tasks in the remaining time. On the other hand, R_J would explore the schedule $T_1; T_2; T_3$, and only after the thread T_3 completes its tasks it will release all three threads again and set the current abstract time to 10. The schedule $T_1; T_2; T_3$ is infeasible because, in a real execution of a program, thread T_1 would be released again before T_2 completes its task and a deadline miss would occur for T_2 and T_3 .

Model of RTSJ. All the new RTSJ-specific components of a program state have to be used in R_J , otherwise it may not explore some feasible execution paths of a real-time Java program and thus might fail to discover some real errors. If some of the RTSJ-specific components were omitted, then R_J might incorrectly decide that two states s_1, s_2 on different state space paths p_1, p_2 , such that $s_1 \in p_1$ and $s_2 \in p_2$, are equivalent, when they actually are not according to the RTSJ semantics, and then it would not explore one of the paths p_1, p_2 beyond the state s_1 , respectively s_2 . Specifically, if the components STQ and RTQ were not added to the program state, then R_J might incorrectly decide that two states s_1 and s_2 , which differ only by the orderings of threads in the queues, are equivalent. If the identification ma of a memory area was not a part of the tuple $do \in DH$, then R_J would not distinguish between the cases in which an object is allocated in different memory areas on different execution paths.

Similarly, all the key RTSJ features have to be implemented in R_J , otherwise it might explore infeasible execution paths of a real-time Java program. If R_J did

```

static class Consumer extends RealtimeThread {
  public void run() {
    while (waitForNextPeriod()) {
      if (buffer.isEmpty()) continue; else buffer.remove();
    }
  }
}
static class Producer extends RealtimeThread {
  public void run() {
    for (int i = 0; i < 100; i++) {
      buffer.add(new String(i));
      if (!waitForNextPeriod()) break;
    }
  }
}

```

Fig. 6. Example program involving spurious data race

not support priority-preemptive scheduling, then it might even report spurious concurrency errors. Given a program in Figure 6 (a modified fragment of the motivating example), in which all threads have the same priority and accesses to the `buffer` variable are not synchronized, JPF without our extensions would explore also execution paths that involve thread preemption at accesses to `buffer` (which is not possible under the RTSJ scheduler) and thus it would report a spurious data race.

A consequence of the clock value not being considered for state matching is that R_J will not loop indefinitely in an attempt to explore an infinite state space path in which some states differ only by clock values. For example, this is the case of the main loop of the `Consumer` thread in Figure 6, where states corresponding to successive calls of `waitForNextPeriod` would differ only by the clock values. R_J will actually explore only one iteration of the loop on any state space path p and then backtrack. This is because (i) a transition is terminated when the `Consumer` thread is put to sleep by calling `waitForNextPeriod` and (ii) two successive states s_1, s_2 on p are equivalent. Were the clock value considered for state matching, then the states s_1 and s_2 would not be equivalent and R_J will attempt to traverse the whole infinite path p .

5 Evaluation

We have implemented R_J as a component of the `RTEEmbed` extension for JPF, which is publicly available in the JPF distribution [10]. In addition to RTSJ-specific errors, such as invalid usage of memory regions, R_J can detect all the general errors (and check the general properties) supported by JPF, such as data races and assertion violations. The tool is completely automatic — support for all the errors and properties is built into the tool, so that a user does not have to define any properties or code annotations by hand. We have evaluated the

proposed approach on CD_x [7], which is a benchmark suite for RTSJ platforms. It is based on a non-trivial multi-threaded program (20 Kloc including libraries) that can be configured and customized to a great degree. Structure of the base program corresponds to the motivating example (Figure 1); it is an instance of the producer-consumer pattern that involves two periodic real-time threads and a main thread.

We have performed experiments on several variants of the CD_x suite. We have used (i) a version of CD_x that does not contain violations of any properties supported by the tool and (ii) and versions of CD_x that contain manually injected errors — we have tried the following three errors: array bounds overflow, invalid assignment of references between memory areas, and a data race. We also performed experiments with the producer-consumer program used as a motivating example. All experiments on CD_x were performed with the same configuration of the suite, in which the producer thread sends 8 messages to the consumer thread. A higher number of messages exchanged between the threads causes an increase of the state space size (due to the increased number of different variable values during the program’s run), but it does not increase the coverage of program’s control-flow paths.

	Time	Memory	States
complete state space of CD_x (no error)	9788 s	1197 MB	118142
producer-consumer (motivating example)	140 s	511 MB	558190
<i>CD_x with manually injected errors</i>			
array bound overflow	22 s	497 MB	12530
invalid assignment	20 s	397 MB	12530
race condition	30 s	412 MB	12574

Table 1. Results of experiments

Results of all the experiments are listed in Table 1. The results show (i) that R_J can explore the complete state space of a non-trivial program in reasonable time, and (ii) that it can be used for efficient detection of errors in multi-threaded real-time Java programs. The source code of R_J , of CD_x , and scripts used to run all experiments is available at: <http://dsrg.mff.cuni.cz/~parizek/rtsjcheck/>.

6 Related work

We are aware only of one existing approach to model checking of RTSJ programs, which is described in [9]. It uses discrete event simulation for modeling of time and discrete event simulation for modeling of scheduling. Unlike our method that aims at detecting time-independent errors related especially to usage of RTSJ memory regions, this approach focuses on checking whether threads in a given RTSJ program meet all deadlines. We see as the main limitation of this approach that it employs an unrealistic model of execution time of bytecode instructions

– a fixed time is defined for each instruction in a look-up table. Some work has been done also on type systems that guarantee correct usage of memory areas by RTSJ programs [2, 12]. In the work described in [2], a type system based on the concepts of ownership and regions is proposed that prevents runtime violations of memory usage rules (e.g., invalid assignments between memory areas). The main limitations of this approach are these: (i) only local variables in methods are supported and (ii) developers have to define some type annotations. On the contrary, our method is completely automatic and it can detect errors involving both instance variables and local variables. The approach described in [12] introduces two code annotations that associate classes with scoped memory areas; nesting of packages determines the nesting of scopes and visibility. While this approach is promising, it requires the developers to follow a particular programming discipline and incurs a minor overhead — code annotations have to be defined by hand. Our method can be used also in the case of RTSJ programs that do not employ the scoped types.

7 Conclusion

In this paper we presented a method for checking multi-threaded real-time Java programs against time-independent safety properties. The key components of our method are (1) an algorithm for state space traversal that follows priority-preemptive scheduling and reflects thread periods, (2) a model of RTSJ semantics and API, and (3) an abstraction of time based on thread periods and release events. We have implemented the method in the R_J tool, which is based on Java PathFinder. Given a multi-threaded program, R_J explores no execution path of the program that involves an infeasible thread interleaving with respect to thread periods and priorities. If checking of a program terminates and no error is reported, then it is guaranteed that the program does not contain any errors detectable by R_J . Nevertheless, due to the unavailability of a precise platform-specific timing model, R_J may explore execution paths that are not possible on a particular target platform. Results of experiments on CD_x show that R_J can be applied for detection of errors in non-trivial multi-threaded RTSJ programs. We plan to improve the precision of the proposed approach by exploiting results of a timing analysis of Java code and knowledge of the history of thread releases on each state space path. We will also look at extending the proposed approach for state space traversal and abstraction of time towards verification of plain Java programs that use relative sleep and timers.

Acknowledgments. This work was partially supported by the Ministry of Education of the Czech Republic (grant MSM0021620838). We also thank Ondrej Sery and Nicholas Kidd for their valuable comments and suggestions.

References

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

2. C. Boyapati, A. Salcianu, W. Beebee, and M. Rinard. Ownership Types for Safe Region-Based Memory Management in Real-Time Java, In Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation (PLDI), ACM, 2003.
3. M. B. Dwyer, J. Hatcliff, M. Hoosier, and Robby. Building Your Own Software Model Checker Using the Bogor Extensible Model Checking Framework, In Proceedings of the 17th International Conference on Computer Aided Verification (CAV 2005), LNCS, vol. 3576, 2005.
4. J. Gosling, B. Joy, G. Steele, and G. Bracha. The Java Language Specification, 3rd Edition, Addison-Wesley, 2005.
5. J. Hunt, D. Locke, K. Nilsen, M. Schoeberl, and J. Vitek. Java for Safety-Critical Applications, In Proceedings of 2nd International Workshop on the Certification of Safety-Critical Software Controlled Systems (SafeCert 2009), to appear in ENTCS.
6. Java PathFinder, <http://babelfish.arc.nasa.gov/trac/jpf/>
7. T. Kalibera, J. Hagelberg, F. Pizlo, A. Plsek, B. Titzer, and J. Vitek. CD_x: A Family of Real-time Java Benchmarks, In Proceedings of the 7th Workshop on Java Technologies for Real-Time and Embedded Systems, ACM, 2009.
8. K.G. Larsen, P. Pettersson, and W. Yi. UPPAAL in a Nutshell, Journal on Software Tools for Technology Transfer (STTT), 1(1-2), 1997.
9. G. Lindstrom, P. C. Mehlitz, and W. Visser. Model Checking Real Time Java Using Java PathFinder, In Proceedings of 3rd International Symposium on Automated Technology for Verification and Analysis (ATVA 2005), LNCS, vol. 3707, 2005.
10. RTEmbed extension for Java PathFinder, <http://babelfish.arc.nasa.gov/trac/jpf/wiki/projects/rtembed>
11. S. Tripakis and C. Courcoubetis. Extending Promela and Spin for Real Time, In Proceedings of the 2nd International Workshop on Tools and Algorithms for Construction and Analysis of Systems (TACAS'96), LNCS, vol. 1055, 1996.
12. T. Zhao, J. Noble, and J. Vitek. Scoped Types for Real-time Java, In Proceedings of the 25th IEEE Real-Time Systems Symposium (RTSS'04), IEEE CS, 2004.