The RTEembed Extension for JPF: Checking Programs for Real-Time and Embedded Systems

Pavel Parízek
DISTRIBUTED SYSTEMS RESEARCH GROUP
http://dsrg.mff.cuni.cz

CHARLES UNIVERSITY PRAGUE
Faculty of Mathematics and Physics
Outline

• Motivation (context)
• Goals

• RTEEmbed extension for JPF
  ▪ Main features
  ▪ Implementation
  ▪ Case study

• Current status
• Future work
Embedded and real-time systems (ERS)

- Mission- or safety-critical systems
  - Errors are very costly → verification is needed
  - Less complex by design → suitable for verification
    - Restrictions: concurrency, memory model, programming, …
  - **Multi-threaded programs**
    - Processing of concurrent inputs (events) from the environment in a timely manner
    - Model checking shown to be a useful verification technique

- Focus on Java software for ERS
  - Java is slowly becoming language of choice for implementation of software for ERS
  - Platforms: **RTSJ, SCJ**
Goals

• Efficient model checking of Java software for embedded and real-time systems
  ▪ Using Java PathFinder model checker (JPF)

• Problems to be solved
  ▪ Model checking of multi-threaded programs is prone to state explosion
  ▪ Implementing RTSJ semantics and API
  ▪ Extending JPF with a notion of time
RTEmbed extension for Java PathFinder

• Main features
  ▪ Platform-specific restrictions of concurrency
    • Joint work with Tomas Kalibera
  ▪ Detector of violations of RTSJ memory usage rules
    • Joint work with Tomas Kalibera, Jan Vitek, Nicholas Kidd, …

• Implementation
• Case study
Platform-specific restrictions of concurrency
Concurrency in Java and ERS

- **Problem**
  - **Java has very permissive model of concurrency**
    - Assumes unlimited number of processors (CPUs, cores)
    - Preemption of threads at any program point (instruction)
    - No restrictions on threading model and scheduling algorithms
  - **Embedded and real-time systems impose platform-specific restrictions on concurrency**
    - Limited number of processors (CPUs, cores), scheduling with limited preemption, …
    - Some thread interleavings allowed by Java semantics cannot happen at runtime

- **Our idea**
  - **Exploiting platform-specific restrictions of concurrency imposed by ERS in model checking with JPF**
    - Benefits: scalability (state explosion), correctness (infeasible paths)
Platform-specific restrictions of concurrency

• Runtime platform for Java programs
  ▪ Hardware: number of processors (CPU, cores)
  ▪ Operating system: scheduling algorithm, thread model
  ▪ JVM: scheduling algorithm, thread model

• Platform configuration
  ▪ Determines the concurrency-related behavior of Java programs running on the platform
Concurrency-related behavior of Java platforms

• Key characteristics of a platform
  ▪ Maximal number of threads that can run in parallel
    • Bounded by the number of processors in HW configuration
    • May be further limited by OS and JVM (e.g. green threading)
  ▪ Threading model and scheduling algorithm
    • Determine the level at which Java threads are scheduled
    • Options: green threading, native threading
  ▪ Set of thread yield points
    • Program points at which a thread can be suspended
      ▪ bytecode instructions, calls of specific method
    • Options: time-based preemption (time slicing), thread yield points at specific instructions
Java platforms for embedded systems

• Key characteristics
  ▪ Single processor is typically available
  ▪ Green threading model
    • Threads are managed and scheduled by JVM
    • All threads are mapped to a single native thread
  ▪ Thread yield points only at specific bytecode instructions and calls of specific methods
    • Synchronization: acquiring or releasing a monitor, calls of methods of the Thread class, calls of wait and notify
    • Other: back-branches, calls of native methods (I/O)
  ▪ Strict enforcement of thread priorities
    • Typical for real-time systems

• Examples
  ▪ CLDC Hotspot (Java ME)
  ▪ Purdue OVM
Java platforms for server and desktop systems

• Key characteristics
  ▪ Multiple processors
  ▪ Native threading model
    • Each Java thread mapped to a native thread
    • Native threads are managed and scheduled by OS
  ▪ Full-fledged time preemption
    • Thread yield points are at all program points

• Examples
  ▪ Sun Java Hotspot
Threading models in current JVMs

Green threads

Java program

Java thread  Java thread  ...  Java thread

scheduler

native process  native process  ...  native process

OS

scheduler

processor  ...  processor

Native threads

Java program

Java thread  Java thread  ...  Java thread

native process  native process  ...  native process

OS

scheduler

processor  ...  processor
Java platforms and restrictions of concurrency

- Java platforms for embedded systems
  - Several restrictions of concurrency → significant reduction of state space size is possible
  - Only some thread scheduling sequences allowed by Java semantics may occur in program’s execution

- Java platforms for server systems
  - Only possible restriction is the number of processors
  - Multiple processors are often used → no significant reduction of state space size is possible
Concurrency and default JPF

• Key characteristics of JPF
  ▪ Adheres to the full Java model of concurrency with no restrictions
    • Unlimited number of processors, scheduling with time preemption
  ▪ Simulates contention for shared data between multiple threads
    • Thread scheduling at bytecode instructions that access shared data or change threads’ status
  ▪ Threads are effectively executed in the interleaved manner
    • Assumes that parallel accesses to the shared memory are actually serialized in HW

• Checking of multi-threaded programs with JPF is inherently prone to state explosion
Checking Java programs with default JPF

```java
public void run() {
    int pos = 0;
    while (true) {
        synchronized (buffer) {
            while (buffer[pos] == null) buffer.wait();
        }

        Object msg = buffer[pos];
        buffer[pos++] = null;
        synchronized (buffer) {
            buffer.notify();
        }
    }
}
```
Restrictions of concurrency and JPF

• Our approach
  ▪ Checking only thread interleavings that may occur at runtime ➔ reduction of the state space size

• Key differences from the default JPF
  ▪ Association of threads to processors
    • Interleavings involving non-active threads are not explored
  ▪ Limited contention for shared data is simulated
    • Passive threads are not scheduled at shared data accesses
Checking Java programs with restrictions and JPF

```java
public void run() {
    int pos = 0;
    while (true) {
        synchronized (buffer) {
            while (buffer[pos] == null) buffer.wait();
        }
        Object msg = buffer[pos];
        buffer[pos++] = null;
        synchronized (buffer) {
            buffer.notify();
        }
    }
}
```

- No thread yield points at accesses to `buffer`: lines 5, 8, and 9
Detecting violations of RTSJ memory usage rules
Overview of RTSJ

- RTSJ is an API for real-time programming in Java
  - Class libraries, constraints on the JVM behavior, and special semantics of some existing language features

- Key features
  - Priority-based thread scheduling
    - FIFO order, periodic releases, priority inversion avoidance
  - Non-garbage collected memory + no-heap threads
    - Scoped memory areas (limited lifetime), immortal memory area
  - Asynchronous transfer of control
    - Interruption of thread's computation at well-defined points
  - Other: asynchronous event handlers, wait-free queues, …

- New runtime errors (exceptions) are introduced
Our goal

• Detecting RTSJ-specific runtime errors
  ▪ Violations of memory usage rules
  ▪ Calls of RT-specific methods by plain Java threads
  ▪ Invalid arguments to RTSJ API methods

• We don’t aim at checking “timeliness properties”
  ▪ No support for RTSJ features and properties that depend on precise model of real time and execution time of instructions on a particular hardware platform
Detecting violations of RTSJ memory usage rules

• Motivation
  ▪ RTSJ memory model is complex and hard to use
    • Writing RTSJ programs is prone to errors
  ▪ Static checker of errors would be very useful

• Example of a memory usage rule
  ▪ Heap and immortal memory areas can never contain references to scoped memory areas (limited lifetime)
    • Dangling references when a scope is reclaimed
Example of a program with error

```java
public class AssignmentError1 {
    // static fields are all in immortal memory
    public static Object fieldInImmortal;

    public static class LogicWithAssignmentError implements Runnable {
        public void run() {
            System.err.println("Running in scope.");

            // assignment error: immortal memory cannot have pointers to scopes
            fieldInImmortal = new Integer(0);
        }
    }

    public static void main( String[] args ) {
        LTMemory scope = new LTMemory( 10000, 10000 );
        RealtimeThread thread = new RealtimeThread(
            null, null, null, scope, null, new LogicWithAssignmentError()
        );
        thread.start();
    }
}
```
Scalability of checking

- State explosion is an issue

- Our solution
  - Exploiting constraints imposed by RTSJ
    - Priority-based FIFO scheduling
    - No preemption by threads with equal priority
    - Ordering of timed events (periods, sleeps, …)
  - Focus on detection of selected errors
Implementation

• RTEmbed extension
  ▪ Part of the official distribution of JPF

• Custom components
  ▪ Thread scheduling (restrictions of concurrency)
    • Choice generators
    • Scheduler factories
    • Implementation of bytecode instructions
  ▪ Model of RTSJ semantics and API
    • Several JPF listeners
    • MJI native peers
Case study

• CDx: Collision Detector
  ▪ Family of real-time Java benchmarks
  ▪ Developed at Purdue University
  ▪ 12 Kloc in Java, three concurrent threads

• Results of experiments
  ▪ Restrictions of concurrency reduce the state space size to a great degree

<table>
<thead>
<tr>
<th>Restrictions</th>
<th>Time (s)</th>
<th>Mem (MB)</th>
<th>States</th>
</tr>
</thead>
<tbody>
<tr>
<td>5 messages</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>default JPF (no restriction)</td>
<td>&gt; 432000</td>
<td>1967</td>
<td>19992569</td>
</tr>
<tr>
<td>quasi-preemptive threading + two processors</td>
<td>2317</td>
<td>1535</td>
<td>102482</td>
</tr>
<tr>
<td>green threading + single processor</td>
<td>127</td>
<td>740</td>
<td>5217</td>
</tr>
<tr>
<td>10 messages</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>default JPF (no restriction)</td>
<td>&gt; 432000</td>
<td>1725</td>
<td>19986412</td>
</tr>
<tr>
<td>quasi-preemptive threading + two processors</td>
<td>41803</td>
<td>2249</td>
<td>2016072</td>
</tr>
<tr>
<td>green threading + single processor</td>
<td>3369</td>
<td>1336</td>
<td>162093</td>
</tr>
</tbody>
</table>

Table 1. Results of experiments on PCD: traversal of the whole state space
Current status

- Restrictions of concurrency
  - Paper accepted at FMICS 2009

- Detector of violations of RTSJ memory usage rules
  - Implementation in progress
    - Advanced RTSJ features (async handlers, etc)
    - Extending JPF with a notion of time
  - Submission expected in winter
Extending JPF with a notion of time

• Why it is important
  ▪ Modeling of periodic execution and sleeps

• Current ideas
  ▪ Non-deterministic wake up
    • Some of the sleeping threads are awoken at each thread scheduling point
  ▪ Partial order of timed events
    • Pruning state space paths that involve infeasible sequences of “wake-up” events
    • Example: T1 with period of 10ms, T2 with period of 100 ms
      ▪ T1 has to wake up ten times between each pair of successive wake-up events for T2
    • Knowledge about real-time thread periods is exploited
      ▪ Designing a platform-independent model of bytecode execution times for model checking with JPF is not realistic
Future work

• Development of Inspector
  ▪ Tool for debugging, monitoring and profiling of verification of a Java program by JPF
  ▪ We plan to use it for debugging of extensions and optimizations for JPF

• Implementing SCJ semantics and API

• Checking components for ERS
  ▪ Modeling and construction of environment with time

• Domain-specific optimizations and heuristics
  ▪ Goal: addressing state explosion even further