Reasoning about Safety Critical Java

Chris Marriott

27\textsuperscript{th} January 2011
Motivation

• Safety critical systems are becoming part of everyday life
  – Failure can potentially lead to serious consequences
• Verification and certification for safety-critical systems [1]
• Formal methods are only practical with tool support [2]

Motivation (2)

• Java has widespread use
  – Large scale programs require high productivity
  – Java has a large sustained community

• Java verification exists [3]

• SCJ
  – Community already exists
  – Industry interested

Java
Reasoning Techniques - Java

- Java PathFinder (JPF)
- Perfect Developer
- Java Modeling Language (JML)
- Extended Static Checking (ESC/Java)
- Java Applet Correctness Kit (Jack)
- Java Interactive Verification Env. (Jive)
- Krakatoa
- KeY System
Java PathFinder [4]

- Translates Java into Promela
- SPIN model checker
- Based on assertions
- Subset of Java
  - Dynamic object creation
  - Inheritance
  - Threads
  - Synchronisation
  - Exceptions
  - Thread Interrupts
Java PathFinder (2)

• Used with real-world NASA examples
• Difficult in large-scale systems
• Error reporting through print statements
• Is the translation optimal?
• Is Promela actually needed?
Perfect Developer [5,6]

• Produces Java, C++ or Ada code from a specification
• Can provide an implementation manually
• Proof obligations created automatically
• Proofs discharged automatically
  – Over 90% success rate for a correct program

• Has been applied to a government IT system [7]
• Verifier not strong enough
  – Code change / additional assertions
• No documentation for the PD architecture
• Specifications for the verifier rather than the program
• Does not support concurrency

JML [8,9,10]

• Behavioural interface specification language

• Code annotations
  – Pre / post conditions
  – Loop invariants

• Tools
  – JML Checker
  – Runtime assertion checking (jmlc & jmlunit)
  – Extended Static Checker (ESC/Java & ESC/Java2)
  – LOOP
    • PVS / Isabelle

Extended Static Checking for Java [11]

- Compile time checking
- Inbetween static checking and formal proofs
- Very successful and popular [12]
- Focused on development
- Neither sound nor complete
- The tool is considered successful if it finds enough errors to repay the cost of running it

Other Tools

• **Jack** [13]
  – Proof obligations for Atelier B, Simplify, and PVS
  – Verification conditions generated per path through code
  – Focused on JavaCARD
  – Similar to LOOP

• **Jive** [14]
  – JML annotations (although not necessary)
  – Interactive theorem prover based on Hoare logic
  – Exceptions not handled

Other Tools (2)

• **Krakatoa** [15]
  - Part of the ‘Why’ software verification platform
  - Does not prove termination of recursive calls
  - Low level of automation for discharging proof obligations

• **KeY** [16]
  - First order dynamic logic
    - *Programs* allowed in pre/post-conditions
    - All Java constructs allowed in the description of states (loops / recursion)
  - Focused on JavaCARD

RTSJ
RTSJ [17]

• Subset of Java
• Designed for real-time systems
• Differences
  – Schedulable objects & pre-emptive priority based scheduling
  – Synchronisation & resource sharing
  – Time values & clock
  – Memory management
  – Asynchronous transfer of control

Reasoning Techniques - RTSJ

• **KeY** [18]
  - JML annotations
  - Translates to dynamic logic
  - Static checking of memory properties

• **Benchmarks \((CD_x)\)** [19]
  - Java, RTSJ, SCJ
  - Exceptions
  - Type analysis
  - Memory analysis
  - Blocking analysis
  - Loop bound analysis

Safety Critical Java [20]

• A subset of RTSJ
• Programming based on missions and event handlers
• Different memory model
  – No heap
  – Immortal memory
  – Scoped memory
• Compliance levels
  – Level 0 – Cyclic executive program
  – Level 1 – Periodic and aperiodic event handlers
  – Level 2 – Real-time no-heap threads
• Current work on reasoning at levels 0 and 1

Reasoning Techniques - SCJ

- Static Checking of Annotations
- Exhaustive Testing
- SafeJML
Static Checking of Annotations [21]

- SCJ annotations
  - Level compliance
  - Behavioural compliance
    - May allocate
    - May self suspend
  - Memory safety
    - Define scope
    - Runs-in scope
- Tool checks implementation against specification
- Are annotations completely necessary?

Exhaustive Testing of SCJ [22]

- JPF extension ($R_{SJ}$)
- Level 0 and 1 support (no APEHs)
  - Memory assignment errors (no need for annotations)
  - Race conditions
  - Priority ceiling emulation violations
  - Dereferencing of null pointers
  - Invalid library calls
  - Array bound violations
  - Divisions by zero
  - Failed assertions
- State explosion when considering APEHs
- “Bug-hunting tool for real-time Java”

SafeJML [23]

• First publically available extension to JML for real-time systems
• Behavioural and timing properties
• Focused on WCET analysis
  – Behaviour properties not considered
• Currently the only specification language for SCJ

## Evaluation

<table>
<thead>
<tr>
<th>Tool</th>
<th>Language</th>
<th>Functional behaviour</th>
<th>Memory properties</th>
<th>Timing properties</th>
</tr>
</thead>
<tbody>
<tr>
<td>JPF</td>
<td>Java</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>Perfect Developer</td>
<td>Java</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>ESC/Java</td>
<td>Java</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>LOOP</td>
<td>Java</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>JACK</td>
<td>Java</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>Jive</td>
<td>Java</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>Krakatoa</td>
<td>Java</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>KeY</td>
<td>RTSJ</td>
<td>No</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>SCJ Annotation Checker</td>
<td>SCJ</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>R$_{SJ}$</td>
<td>SCJ</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>SafeJML</td>
<td>SCJ</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
</tr>
</tbody>
</table>
Conclusion

• Lots of tools/techniques for Java
• Less for RTSJ and SCJ
• Currently no way to verify functional behaviour for SCJ level 1
• Model checking can lead to state explosion
• Theorem proving is difficult to automate
Open Questions

• Can we apply existing tools and techniques to SCJ?

• Can we extend these techniques?

• Are SCJ annotations necessary and sufficient?