Enabling the Runtime Assertion Checking of Concurrent Contracts for the Java Modeling Language
by Wladimir Araujo, Lionel C. Briand, Yvan Labiche

CCC Seminar Talk

Alexander Kogtenkov

ETH Zürich

April 15, 2015


Wladimir Araujo
Juniper Networks

Lionel C. Briand
Simula Research Laboratory and University of Oslo

Yvan Labiche
Carleton University
Design by Contract in a nutshell
Theory: correct program + DbC ⇔ correct program
Run-time:
  - Sequential: regular ≡ instrumented
  - Concurrent: execution time affects execution paths

How to restore equivalence?
Concurrent DbC  Challenges

- Interference
  
  Objects involved in contracts may be changed before/after or during execution of a method

- Locking-related properties
  
  Deadlocks

- Specification of thread-safety properties in presence of inheritance
  
  Thread-safety predicates may become meaningless in descendants
Interference   Sequential JML

Method source code

Byte code

Precondition

Body

Postcondition

Instrumented byte code

call precondition
call body
call postcondition

Precondition       Evaluation       Postcondition

A. Kogtenkov  CCC seminar: Concurrent contracts for Java  SS 2015
Interference  

**Example: Linked queue**

```java
/* @ public normal_behavior
3 @ ensures \result \iff head.next == null */
public boolean isEmpty () {
1 synchronized (head) {
2 return head.next == null;
}}
```

<table>
<thead>
<tr>
<th></th>
<th></th>
<th>&quot;make list empty&quot;</th>
</tr>
</thead>
<tbody>
<tr>
<td>T1</td>
<td>1’</td>
<td>lock (head)</td>
</tr>
<tr>
<td>T1</td>
<td>2</td>
<td>result = (head.next == null)</td>
</tr>
<tr>
<td>T1</td>
<td>1”</td>
<td>unlock (head)</td>
</tr>
<tr>
<td>T2</td>
<td></td>
<td>insert (v)</td>
</tr>
<tr>
<td>T1</td>
<td>3</td>
<td>result \iff head.next == null</td>
</tr>
</tbody>
</table>

Postcondition violation!
Interference  

**Example: Linked queue**

```java
/* @ public normal_behavior
3 @ ensures \result \iff head.next == null */

public boolean isEmpty () {
1  synchronized (head) {
   // @ ensures_safepoint:
   // @ ensures_safepoint:
   // @ ensures_safepoint:
   2 return head.next == null;
}
```

---

<table>
<thead>
<tr>
<th></th>
<th></th>
<th>”make list empty“</th>
<th>head.next == null</th>
</tr>
</thead>
<tbody>
<tr>
<td>T1</td>
<td>1'</td>
<td>lock (head)</td>
<td>[head]</td>
</tr>
<tr>
<td>T1</td>
<td>2</td>
<td>result =</td>
<td>[head],</td>
</tr>
<tr>
<td></td>
<td></td>
<td>(head.next == null)</td>
<td>result == true</td>
</tr>
<tr>
<td>T1</td>
<td>1&quot;</td>
<td>unlock (head)</td>
<td>result == true</td>
</tr>
<tr>
<td>T2</td>
<td>...</td>
<td>insert (v)</td>
<td></td>
</tr>
<tr>
<td>T1</td>
<td>3</td>
<td>result \iff</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>head.next == null</td>
<td></td>
</tr>
</tbody>
</table>
```

A. Kogtenkov  

CCC seminar: Concurrent contracts for Java  

SS 2015  

7 / 14
/* @ public normal_behavior
3  @ ensures \result <==> head.next == null */

public boolean isEmpty () {
1  synchronized (head) {
    // @ ensures_safepoint:

2  return head.next == null;
}
}

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>head.next == null</td>
</tr>
<tr>
<td>T1</td>
<td>1'</td>
<td>lock (head)</td>
<td>[head]</td>
</tr>
<tr>
<td>T1</td>
<td>2</td>
<td>result = (head.next == null)</td>
<td>[head], result == true</td>
</tr>
<tr>
<td>T1</td>
<td>3</td>
<td>result &lt;==&gt; head.next == null</td>
<td>[head], Postcondition OK</td>
</tr>
<tr>
<td>T1</td>
<td>1&quot;</td>
<td>unlock (head)</td>
<td>result == true</td>
</tr>
</tbody>
</table>
Interference

Concurrent JML

Sequential translation

time

Precondition  sync  Evaluation  sync  Postcondition

Body

Concurrent translation

time

sync  Precondition  Evaluation  Postcondition  sync
Interference

Other mechanisms

<table>
<thead>
<tr>
<th>Mechanism</th>
<th>Specification</th>
<th>Implementation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Safepoints</td>
<td>requires ensures</td>
<td>requires_safepoint ensures_safepoint</td>
</tr>
<tr>
<td>Wait condition</td>
<td>when</td>
<td>commit</td>
</tr>
<tr>
<td>Thread safety</td>
<td>requires_thread_safe ensures_thread_safe</td>
<td>__</td>
</tr>
</tbody>
</table>
### Other mechanisms

<table>
<thead>
<tr>
<th>Mechanism</th>
<th>Specification</th>
<th>Implementation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Safepoints</td>
<td>requires ensures</td>
<td>requires_safepoint ensures_safepoint</td>
</tr>
<tr>
<td>Wait condition</td>
<td>when</td>
<td>commit</td>
</tr>
<tr>
<td>Thread safety</td>
<td>requires_thread_safe ensuring_thread_safe</td>
<td>__</td>
</tr>
</tbody>
</table>

**Scope**

- Safepoints: internal state
- Thread safety: external state

**Thread safety + Safepoints = No Interference**
Deadlock avoidance

Lock order specification

Syntax

lock_order lock1 < lock2
lock_order lock1 <= lock2

Semantics

<table>
<thead>
<tr>
<th>Acquisition order</th>
<th>&lt;</th>
<th>&lt;=</th>
</tr>
</thead>
<tbody>
<tr>
<td>lock1 before lock2</td>
<td>true</td>
<td>true</td>
</tr>
<tr>
<td>lock1 only</td>
<td>true</td>
<td>true</td>
</tr>
<tr>
<td>lock2 before lock1</td>
<td>false</td>
<td>false</td>
</tr>
<tr>
<td>lock2 only</td>
<td>false</td>
<td>false</td>
</tr>
<tr>
<td>neither lock1 not lock2</td>
<td>false</td>
<td>true</td>
</tr>
</tbody>
</table>

Monitoring

At every lock acquisition point
Before actually attempting to acquire a lock (to prevent deadlocks)
Equivalence assessment  Run-time overhead

- Setup: Router driver of Juniper’s E-series routers, 520 000 active subscribers, 1 500 transactions per second
  - 54 classes
  - 33 509 LOC
  - 34% concurrent behavior

- Measurement
  
  Performance ratio between production and instrumented versions is **constant**.

  - Heap consumption: 3.47
  - CPU load: 17.5
  - Original
  - Instrumented

A. Kogtenkov  CCC seminar: Concurrent contracts for Java  SS 2015
Equivalence assessment  Indistinguishability

Analysis of behavior:

- Standard test suite
  - 2 hours, no contract violations
- Faults introduced from Juniper’s defect database
  - Instrumentation to print errors instead of assertion violations
  - Reproduced all 139 functional and concurrent faults
- Influence of instrumentation on thread interleavings
  - No additional locks
    - Safepoint evaluation (inside synchronized)
    - Commit points for wait conditions (inside synchronized)
    - Locking predicates (use thread-local objects in pre- and post-states)
- Limited locking
  - ConcurrentHashMap optimized for concurrent access
  - Writes only in pre- and post-states of methods with requires / ensures thread safety
  - Reads rarely acquire locks
## Beyond the paper

### Comparison to SCOOP

<table>
<thead>
<tr>
<th>Property</th>
<th>Concurrent JML</th>
<th>SCOOP</th>
</tr>
</thead>
<tbody>
<tr>
<td>Syntax</td>
<td>&gt; 10 keywords</td>
<td>separate</td>
</tr>
<tr>
<td>Internal state</td>
<td>Safepoints</td>
<td>No data races</td>
</tr>
<tr>
<td>External state</td>
<td>Thread-safety</td>
<td>Controlled arguments</td>
</tr>
<tr>
<td>Wait conditions</td>
<td>Explicit</td>
<td>Separate calls in</td>
</tr>
<tr>
<td></td>
<td></td>
<td>preconditions</td>
</tr>
<tr>
<td>Synchronization</td>
<td>Explicit locking</td>
<td>Controlled arguments</td>
</tr>
<tr>
<td>Deadlock avoidance</td>
<td>Lock order</td>
<td>Partial: controlled</td>
</tr>
<tr>
<td></td>
<td></td>
<td>arguments</td>
</tr>
</tbody>
</table>
Beyond the paper  Open questions

For further discussion

• What is allowed between method entry and requires_safe_point?
• Blocks concurrent_behavior are combined using conjunction. What about modularity?
• What feature of concurrent JML would be nice to have in SCOOP?
• What can be improved or simplified in concurrent JML?
• Does lock order specification prevent from deadlocks?