IMUnit: Improved Multithreaded Unit Testing

Vilas Jagannath, Milos Gligoric, Dongyun Jin, Qingzhou Luo
Grigore Rosu, Darko Marinov

April 14\textsuperscript{th} 2011
UPCRC Seminar
Multicore World

• Parallel code required for performance
  – Dominant paradigm is shared-memory, multithreaded code

• Difficult to develop (correct) multithreaded code
  – Different behavior under different schedules
  – Afflicted by bugs like data races, deadlocks, atomicity violations…

• Difficult to test multithreaded code
  – Bugs triggered by specific schedules
  – Need to specify schedules in tests
  – Exploration required, time consuming
Unit Testing Multithreaded Code

• How to **write** multithreaded unit tests?
  – Bugs are dependent on schedule
  – How to **express schedules** in unit tests?

• How to **explore** multithreaded unit tests?
  – Current techniques focus on one code version
  – Code evolves, need **efficient regression testing**

• How to **generate** multithreaded unit tests?
  – How to **automatically generate test code**?
  – How to **automatically generate schedules**?
IMUnit

• Writing multithreaded unit tests (today’s talk)
  – New language for expressing schedules
  – Automated migration of legacy unit tests
  – Monitor-based enforcement of schedules

• Regression testing
  – Selecting/prioritizing exploration of change-impacted schedules

• Generating tests
  – Generating schedules prior work, generating code in progress
Example: ArrayBlockingQueue

- Array-backed implementation of a bounded blocking queue
  - Provided by java.util.concurrent

  ![ArrayBlockingQueue Diagram]

- **add** operation
  - Inserts into tail of queue
  - Throws exception if queue is full

- **take** operation
  - Removes and returns head of queue
  - Blocks if queue is empty

- Want to test operations from multiple threads
Traditional, Sleep-Based Test

```java
public void testTakeWithAdd() {
    ...
    q = new ArrayBlockingQueue<Integer>(1);
    Thread addThread = new Thread(
        new CheckedRunnable() {
            public void realRun() {
                q.add(1);
                Thread.sleep(150);
                q.add(2);
            }
        }, "addThread").start();
    Thread.sleep(50);
    Integer taken = q.take();
    assertTrue(taken == 1 && q.isEmpty());
    taken = q.take();
    assertTrue(taken == 2 && q.isEmpty());
    ...
}
```

Sleeps used to express and enforce schedules

What schedule is tested in this example?
public void testTakeWithAdd() {
    ...
    q = new ArrayBlockingQueue<Integer>(1);
    Thread addThread = new Thread(
        new CheckedRunnable() {
            public void realRun() {
                q.add(1);
                Thread.sleep(150);
                q.add(2);
            }
        }, "addThread").start();
    Thread.sleep(50);
    Integer taken = q.take();
    assertTrue(taken == 1 && q.isEmpty());
    taken = q.take();
    assertTrue(taken == 2 && q.isEmpty());
    ...
}
Example Schedule (1)

- Testing operations from multiple threads:
  - add followed by a non-blocking take
public void testTakeWithAdd() {
    ...
    q = new ArrayBlockingQueue<Integer>(1);
    Thread addThread = new Thread(new CheckedRunnable() {
        public void realRun() {
            q.add(1);
            Thread.sleep(150);
            q.add(2);
        }
    }, "addThread").start();
    Thread.sleep(50);
    Integer taken = q.take();
    assertTrue(taken == 1 && q.isEmpty());
    taken = q.take();
    assertTrue(taken == 2 && q.isEmpty());
    ...
}
Example Schedule (2)

- Testing operations from multiple threads:
  - *add* followed by a non-blocking *take*
  - blocking *take* followed by an *add*

100s of such schedules in TCK for java.util.concurrent!
Sleep-Based Tests: Issues

public void testTakeWithAdd() {
    ...
    q = new ArrayBlockingQueue<Integer>(1);
    Thread addThread = new Thread(
        new CheckedRunnable() {
            public void realRun() {
                q.add(1);
                Thread.sleep(150);
                q.add(2);
            }
        }, "addThread").start();
    Thread.sleep(50);
    Integer taken = q.take();
    assertTrue(taken == 1 && q.isEmpty());
    taken = q.take();
    assertTrue(taken == 2 && q.isEmpty());
    ...
}
MultithreadedTC: Tick-Based Tests

• Other researchers have noticed sleep-based test issues
• Latest solution: MultithreadedTC
  – Bill Pugh and Nathaniel Ayewah (University of Maryland)
MultithreadedTC: Tick-Based Tests

```java
public class TestTakeWithAdd extends MultithreadedTest {
    ...
    public void initialize() {
        q = new ArrayBlockingQueue<Integer>(1);
    }
    public void addThread() {
        q.add(1);
        waitForTick(2);
        q.add(2);
    }
    public void takeThread() {
        waitForTick(1);
        Integer taken = q.take();
        assertTrue(taken == 1 && q.isEmpty());
        taken = q.take();
        assertTick(2);
        assertTrue(taken == 2 && q.isEmpty());
    }
}
```

Global logical clock ticks used to specify schedules

- `waitForTick`: blocks thread until tick is reached
- `assertTick`: asserts against current clock value
- `Clock incremented when all threads are blocked`
public class TestTakeWithAdd extends MultithreadedTest {

    ...  
    public void initialize() {
        q = new ArrayBlockingQueue<Integer>(1);
    }
    public void addThread() {
        q.add(1);
        waitForTick(2);
        q.add(2);
    }
    public void takeThread() {
        waitForTick(1);
        Integer taken = q.take();
        assertTrue(taken == 1 && q.isEmpty());
        taken = q.take();
        assertTick(2);
        assertTrue(taken == 2 && q.isEmpty());
    }
}

q.add(1) before q.take()
public class TestTakeWithAdd extends MultithreadedTest {

... public void initialize() {
    q = new ArrayBlockingQueue<Integer>(1);
}

public void addThread() {
    q.add(1);
    waitForTick(2);
    q.add(2);
}

public void takeThread() {
    waitForTick(1);
    Integer taken = q.take();
    assertTrue(taken == 1 && q.isEmpty());
    taken = q.take();
    assertTick(2);
    assertTrue(taken == 2 && q.isEmpty());
}
}
MultithreadedTC: Features

```java
public class TestTakeWithAdd extends MultithreadedTest {
    ...
    public void initialize() {
        q = new ArrayBlockingQueue<Integer>(1);
    }
    public void addThread() {
        q.add(1);
        waitForTick(2);
        q.add(2);
    }
    public void takeThread() {
        waitForTick(1);
        Integer taken = q.take();
        assertTrue(taken == 1 && q.isEmpty());
        taken = q.take();
        assertTick(2);
        assertTrue(taken == 2 && q.isEmpty());
    }
}
```

+ Reliable, unlike sleeps
- Not intuitive to reason about ticks
- Depart greatly from sleep-based JUnit tests
Sleep-Based and Tick-Based Tests

• Sleep-Based
  – Unreliable
  – Inefficient
  – Unintuitive

• Tick-Based
  + Reliable
  – Still unintuitive
  – Depart greatly from sleep-based tests

• Need a reliable, efficient, intuitive, easy to migrate solution

• IMUnit: Event-based tests
IMUnit: Event-Based Tests

```
@Schedule("afterAdd1->beforeTake1, [beforeTake2]->beforeAdd2")
public void testTakeWithAdd() {
  ...
  q = new ArrayBlockingQueue<Integer>(1);
  Thread addThread = new Thread(
    new CheckedRunnable() {
      public void realRun() {
        q.add(1);
        @Event("afterAdd1")
        @Event("beforeAdd2")
        q.add(2);
      }
    }, "addThread").start();
  @Event("beforeTake1")
  Integer taken = q.take();
  assertTrue(taken == 1 && q.isEmpty());
  @Event("beforeTake2")
  taken = q.take();
  assertTrue(taken == 2 && q.isEmpty());
  ...
```

Event ordering constraints used to specify schedules
@Schedule("afterAdd1->beforeTake1, [beforeTake2]->beforeAdd2")
public void testTakeWithAdd() {
...
q = new ArrayBlockingQueue<Integer>(1);
Thread addThread = new Thread(
    new CheckedRunnable() {
        public void realRun() {
            q.add(1);
            @Event("afterAdd1")
            @Event("beforeAdd2")
            q.add(2);
        }
    }, "addThread").start();
@Event("beforeTake1")
Integer taken = q.take();
assertTrue(taken == 1 && q.isEmpty());
@Event("beforeTake2")
taken = q.take();
assertTrue(taken == 2 && q.isEmpty());
...
Schedule Annotations

```java
@Schedule("afterAdd1->beforeTake1, [beforeTake2]->beforeAdd2")
public void testTakeWithAdd() {
    ...
    q = new ArrayBlockingQueue<Integer>(1);
    Thread addThread = new Thread(
        new CheckedRunnable() {
            public void realRun() {
                q.add(1);
                @Event("afterAdd1")
                @Event("beforeAdd2")
                q.add(2);
            }
        }, "addThread").start();
    @Event("beforeTake1")
    Integer taken = q.take();
    assertTrue(taken == 1 && q.isEmpty());
    @Event("beforeTake2")
taken = q.take();
    assertTrue(taken == 2 && q.isEmpty());
    ...
}
```

@Schedule: partial ordering between events

\[ e \rightarrow e' \equiv e \text{ before } e' \]

[e] \equiv e \text{ happens then thread blocks}
IMUnit: Features

@Schedule("afterAdd1->beforeTake1, [beforeTake2]->beforeAdd2")
public void testTakeWithAdd() {
...
q = new ArrayBlockingQueue<Integer>(1);
Thread addThread = new Thread(
    new CheckedRunnable() {
        public void realRun() {
            q.add(1);
            @Event("afterAdd1")
            @Event("beforeAdd2")
            q.add(2);
        }
    }, "addThread").start();
@Event("beforeTake1")
Integer taken = q.take();
assertTrue(taken == 1 && q.isEmpty());
@Event("beforeTake2")
taken = q.take();
assertTrue(taken == 2 && q.isEmpty());
...
public void testTakeWithAdd() {
    ...
    q = new ArrayBlockingQueue<Integer>(1);
    Thread addThread = new Thread(
        new CheckedRunnable() {
            public void realRun() {
                q.add(1);
                Thread.sleep(150);
                q.add(2);
            }
        }, "addThread").start();
    Thread.sleep(50);
    Integer taken = q.take();
    assertTrue(taken == 1 && q.isEmpty());
    taken = q.take();
    assertTrue(taken == 2 && q.isEmpty());
    ...
}
IMUnit Schedule Language

<Event Name> ::= { <Id> "." } <Id>
.Thread Name> ::= <Id>
.Basic Event> ::= <Event Name> ["@" <Thread Name>] | "start" "@" <Thread Name>
 | "end" "@" <Thread Name>
<Block Event> ::= "[" <Basic Event> "]"
<Condition> ::= <Basic Event> | <Block Event> | <Condition> "||" <Condition>
 | <Condition> "&&" <Condition> | "(" <Condition> ")"
<Ordering> ::= <Condition> "->" <Basic Event>
<Schedule> ::= { <Ordering> [","] }

- Events:
  - Two types: non-blocking-event and [blocking-event]
  - Can be parameterized by thread-name: event@threadName
  - Can also be combined into conditions using "||" and "&&"

- Ordering specifies order between a condition and event
  - "->" is the ordering operator
  - before-condition -> after-event

- Schedule is a comma-separated list of orderings
Underlying Schedule Logic

• Fragment of PTLTL
  – Over finite well formed multithreaded unit test execution traces
  – Two temporal operators
    • Block
    • Ordering

• Guided by practical requirements
  – Over 200 existing multithreaded unit tests

• Details in paper (under submission)

Logic Syntax:

\[ a ::= \text{start} | \text{end} | \text{block} | \text{unblock} | \text{event names} \]
\[ t ::= \text{thread names} \]
\[ e ::= a@t \]
\[ \varphi ::= [t] \mid \varphi \rightarrow \varphi \mid \text{usual propositional connectives} \]

Logic Semantics:

The semantics of our logic is defined as follows:
\[ e_1e_2\ldots e_n \vDash e \text{ iff } e = e_n \]
\[ \tau \vDash \varphi \land \lor \psi \text{ iff } \tau \vDash \varphi \text{ and/or } \tau \vDash \psi \]
\[ e_1e_2\ldots e_n \vDash [t] \text{ iff } (\exists 1 \leq i \leq n) (e_1 = \text{block}@t \text{ and } (\forall j < i \leq n) e_j \neq \text{unblock}@t) \]
\[ e_1e_2\ldots e_n \vDash \varphi \rightarrow \psi \text{ iff } (\forall 1 \leq i \leq n) e_1e_2\ldots e_i \neq \psi \text{ or } (\exists 1 \leq i \leq n) (e_1e_2\ldots e_i = \psi \text{ and } (\exists j \geq i) e_{i+1}e_{i+2}\ldots e_j = \varphi) \]

It is not hard to see that the two new operators \([t]\) and \(\varphi \rightarrow \psi\) can be expressed in terms of PTLTL as
\[ [t] \equiv \neg \text{unblock}@t S \text{ block}@t \]
\[ \varphi \rightarrow \psi \equiv \Box \neg \psi \lor \Diamond (\psi \land \Diamond \varphi) \]

where \(S\) stands for “since” and \(\Box\) for “always in the past”.

25
Manual Migration

- We manually migrated over 200 sleep-based tests to IMUnit
- Migration typically involved the following steps:
  1. Optionally name threads (default names non-deterministic)
  2. Introduce events using @Event annotations
     - Need to identify interesting points
  3. Introduce schedule using @Schedule annotation
     - Need to understand intended sleep-based schedule
     - Specify the orderings required by intended schedule
     - Also identify blocking vs. non-blocking events
  4. Check that added schedule is the intended schedule
  5. Remove sleeps
  6. Optionally merge tests with different schedules but similar code

- **Automated first four steps**
Automated Migration

• Introducing events and schedules most challenging
• Likely events and schedules inferred from execution logs
• Two phase process
• Automation implemented as an Eclipse refactoring plugin
Logging

• Logs collected from N passing runs

• Logging specific to inference
  – Event inf: original sleep-based test
  – Schedule inf: sleep-based test with events (automatically inferred or manually written)

• Lightweight logging - minimal perturbation

• Only relevant operations logged
  – Thread start, end
  – Sleep call, return
  – Blocking call, return (wait, park)
  – Event call, return (for schedule inference)
  – Other method call, return (for event inference)
Events Inference

- Infer likely location and name of events
- Intuition – sleeps force completion of operations in other threads
- Each log split into regions between sleep call and return
- Regions used to infer “after” events in non-sleeping threads
- “before” event is inferred from the sleeping thread
- Low confidence events removed
Schedule Inference

- Infers likely orderings between events
- Intuition – sleeps induce orderings between sleeping and other threads
- Events found by scanning log around a sleep call
- Block call identifies blocking event
- Low confidence orderings removed
- Additional filtering
Refactoring Plugin

- Naming threads or warning if thread is not named
- Events inference and insertion of annotations
- Schedule inference and insertion of annotation
- Checking inferred schedule is intended schedule
Schedule Enforcement & Checking

- Implemented using JavaMOP
- Schedule logic implemented as a JavaMOP logic plugin
- Takes as input a schedule and outputs a monitor
- Java-shell for schedule language converts monitor into aspects that are weaved into test code
- Different monitor for each test, schedule pair
- Monitor can work in two modes:
  - Active mode enforces schedules
  - Passive mode prints error if execution deviates from schedule
Example Monitor Pseudocode

Generated Monitor:

```java
switch (event) {
    case afterAdd1:
        occurred_afterAdd1 = true; wakeAll();
        break;
    case beforeTake2:
        thread_beforeTake2 = currentThread();
        occurred_beforeTake2 = true; wakeAll();
        break;
    case beforeTake1:
        while(!(occurred_afterAdd1))
            wait();
        occurred_beforeTake1 = true; wakeAll();
        break;
    case beforeAdd2:
        while(!(occurred_beforeTake2 &&
                blocked(thread_beforeTake2)))
            wait();
        occurred_beforeAdd2 = true; wakeAll();
        break;
}
```
Evaluation

• Expressiveness of schedule language
• Precision and recall for event and schedule inference
• Efficiency of schedule enforcement
Expressiveness of Schedule Language

- Experience with migrating over 200 sleep-based unit tests
  - 7 different open source projects
- Evolved language using migration experience
  - Blocking events added because they were required by many tests
  - Events in loops were only required for 5 tests so not added
- Replaced sleeps with events and schedules in 198 tests

<table>
<thead>
<tr>
<th>Subject</th>
<th>Description</th>
<th>Classes</th>
<th>Testcases</th>
</tr>
</thead>
<tbody>
<tr>
<td>Collections</td>
<td>Apache Common Collections</td>
<td>2</td>
<td>18</td>
</tr>
<tr>
<td>JBoss-Cache</td>
<td>JBoss Cache Server</td>
<td>7</td>
<td>27</td>
</tr>
<tr>
<td>Lucene</td>
<td>Apache Lucene</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>Mina</td>
<td>Apache Mina</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Pool</td>
<td>Apache Common Pool</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>Sysunit</td>
<td>Junit extension</td>
<td>5</td>
<td>9</td>
</tr>
<tr>
<td>TCK</td>
<td>JSR-166 TCK</td>
<td>18</td>
<td>139</td>
</tr>
</tbody>
</table>
Precision and Recall for Inference

- Compared automated migration with manual migration
- Automatically inferred events vs. Manually added events
  - Compared location only, not event names
- Automatically inferred orderings vs. Manually added orderings
  - Schedule inferred using manually added (not inferred) events

<table>
<thead>
<tr>
<th>Subject</th>
<th>Inferring Events</th>
<th>Inferring Schedules</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Precision</td>
<td>Recall</td>
</tr>
<tr>
<td>Collections</td>
<td>0.76</td>
<td>0.76</td>
</tr>
<tr>
<td>JBoss-Cache</td>
<td>0.81</td>
<td>0.84</td>
</tr>
<tr>
<td>Lucene</td>
<td>0.33</td>
<td>0.33</td>
</tr>
<tr>
<td>Mina</td>
<td>1.00</td>
<td>1.00</td>
</tr>
<tr>
<td>Pool</td>
<td>0.90</td>
<td>1.00</td>
</tr>
<tr>
<td>Sysunit</td>
<td>0.78</td>
<td>0.83</td>
</tr>
<tr>
<td>TCK</td>
<td>0.69</td>
<td>0.75</td>
</tr>
<tr>
<td>Arithmetic Mean</td>
<td>0.75</td>
<td>0.79</td>
</tr>
</tbody>
</table>
# Efficiency of Schedule Enforcement

- IMUnit test execution vs. sleep-based test execution
  - IMUnit test execution more than 3x faster
    - Schedule enforcement is efficient
- Also demonstrates the over estimation of sleep delays
  - Sleeps are inefficient

<table>
<thead>
<tr>
<th>Subject</th>
<th>Original [s]</th>
<th>IMUnit [s]</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>Collections</td>
<td>5.00</td>
<td>0.89</td>
<td>5.62</td>
</tr>
<tr>
<td>JBoss-Cache</td>
<td>66.53</td>
<td>31.80</td>
<td>2.09</td>
</tr>
<tr>
<td>Lucene</td>
<td>11.67</td>
<td>3.90</td>
<td>2.99</td>
</tr>
<tr>
<td>Mina</td>
<td>0.29</td>
<td>0.17</td>
<td>1.71</td>
</tr>
<tr>
<td>Pool</td>
<td>1.43</td>
<td>1.04</td>
<td>1.38</td>
</tr>
<tr>
<td>Sysunit</td>
<td>17.67</td>
<td>0.41</td>
<td>43.10</td>
</tr>
<tr>
<td>TCK</td>
<td>15.16</td>
<td>10.20</td>
<td>1.49</td>
</tr>
<tr>
<td><strong>Geometric Mean</strong></td>
<td></td>
<td></td>
<td><strong>3.41</strong></td>
</tr>
</tbody>
</table>

- IMUnit test execution more than 3x faster
- Schedule enforcement is efficient
- Also demonstrates the over estimation of sleep delays
  - Sleeps are inefficient
Summary

- Writing multithreaded unit tests is difficult
  - Need to specify schedules
- Current solutions for specifying schedules
  - Sleep-based: unreliable, inefficient, unintuitive, non-modular
  - Tick-based: unintuitive, non-modular
- IMUnit:
  - Explicit event-based schedule language
  - Expressed using simple annotations
  - Reliable, intuitive, modular, efficient
  - Automated migration – events and schedule inference
  - Monitor based schedule enforcement & checking
- Schedule language is expressive
- Inference has good precision and recall
- Schedule enforcement is efficient