IMUnit:
Improved Multithreaded Unit Testing

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

ILLINOIS
UNIVERSITY OF ILLINOIS AT URBANA-CHAMPAIGN

April 14th 2011
UPCRC Seminar

UPCRC Illinois
Universal Parallel Computing Research Center

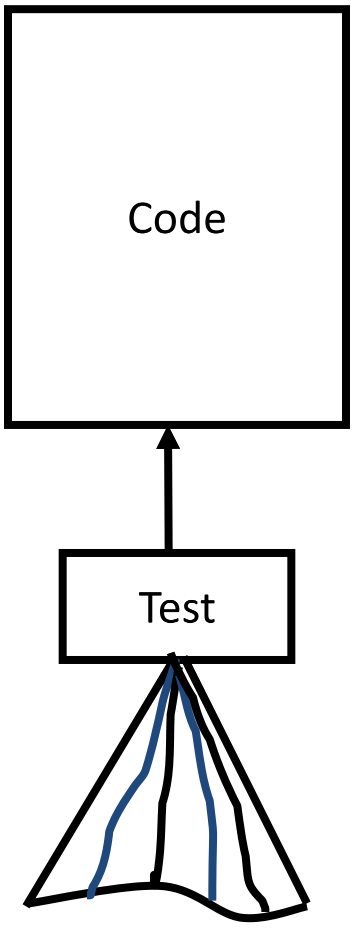
Multicore World

- 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

UPCRC Illinois
Universal Parallel Computing Research Center

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**?



UPCRC Illinois
Universal Parallel Computing Research Center

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

- ***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

UPCRC Illinois
Universal Parallel Computing Research Center

Traditional, Sleep-Based Test

```
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?

Two Sleeps (1)

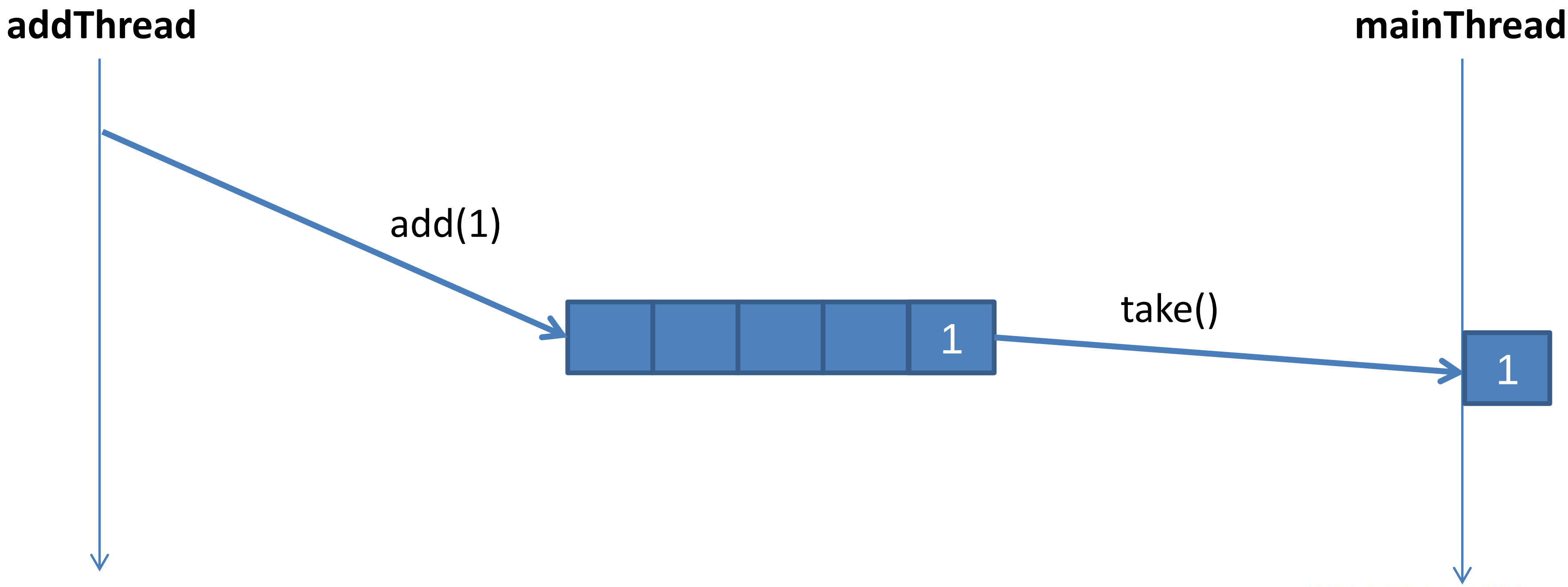
```java
public void testTakeWithAdd() {
  …
  q = new ArrayBlockingQueue<Integer>(1);
  Thread addThread = new Thread(
    new CheckedRunnable() {
      public void realRun() {
        q.add(1);
        Thread.sleep(150);
        q.add(2);
      }
    }, "addThread").start();
  Thread.sleep(50);
  Integer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

Thread.sleep(50):

q.add(1) before q.take()

Example Schedule (1)

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



UPCRC Illinois
Universal Parallel Computing Research Center

Two Sleeps (2)

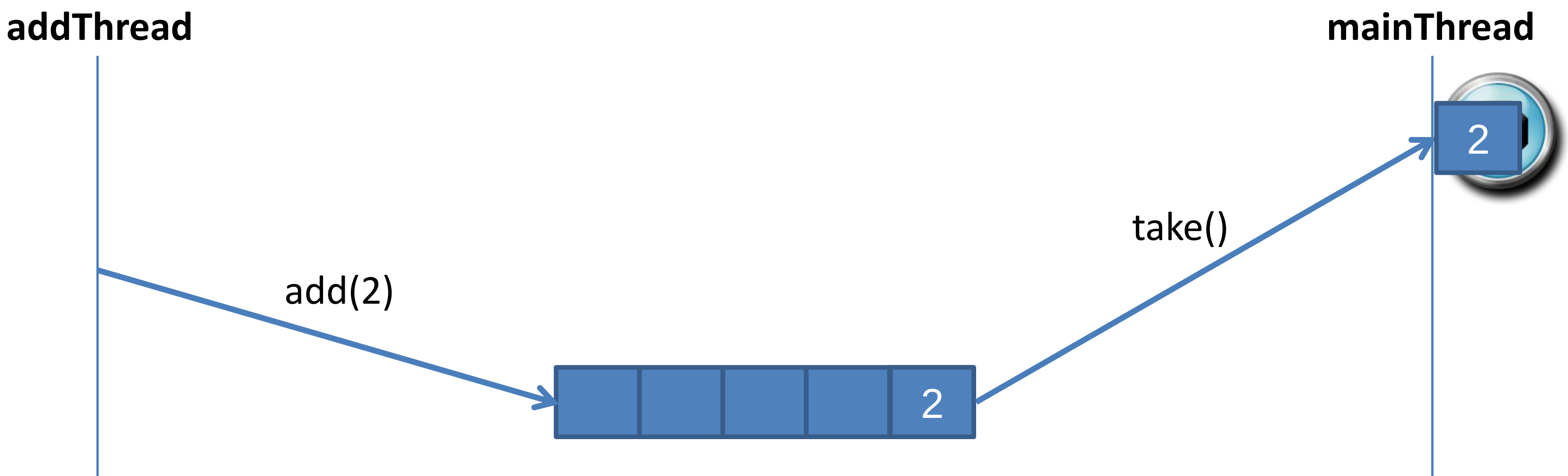
```
public void testTakeWithAdd() {
  …
  q = new ArrayBlockingQueue<Integer>(1);
  Thread addThread = new Thread(
    new CheckedRunnable() {
      public void realRun() {
        q.add(1);
        Thread.sleep(150);
        q.add(2);
      }
    }, "addThread").start();
  Thread.sleep(50);
  Integer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

Thread.sleep(150):
q.take() blocks, then
q.add(2) happens

UPCRC Illinois
Universal Parallel Computing
Research Center

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());
  …
}
```

- **Unreliable!**
  - **False positives/negatives**
- **Inefficient**
  - **Overestimated delays**
- **Unintuitive**

UPCRC Illinois
Universal Parallel Computing Research Center

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

```
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

Clock incremented when all threads are blocked

assertTick: asserts against current clock value

UPCRC Illinois
Universal Parallel Computing Research Center

MultithreadedTC: Two Schedules (1)

```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());
  }
}
```

q.add(1) before q.take()

MultithreadedTC: Two Schedules (2)

```
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.take() blocks, then q.add(2) happens

MultithreadedTC: Features

```
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**

UPCRC Illinois
Universal Parallel Computing Research Center

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

Event Annotations

```
@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: interesting point in execution of a thread

UPCRC Illinois
Universal Parallel Computing Research Center

Schedule Annotations

```
@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->e' ≡ e before e'

[e] ≡ e happens then thread blocks

UPCRC Illinois
Universal Parallel Computing Research Center

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());
  …
```

+ Reliable
+ Intuitive?
+ Modular
+ Similar to sleep-based JUnit tests

Sleep-Based vs. Event-Based

```
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());
  …
}
```

```
@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());
  …
}
```

UPCRC Illinois
Universal Parallel Computing Research Center

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

UPCRC Illinois
Universal Parallel Computing Research Center

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:

$$
\begin{array}{lll}
a & ::= & \mathit{start} \mid \mathit{end} \mid \mathit{block} \mid \mathit{unblock} \mid \text{event names} \\
t & ::= & \text{thread names} \\
e & ::= & a@t \\
\varphi & ::= & [t] \quad \mid \quad \varphi \rightarrow \varphi \quad \mid \quad \text{usual propositional connectives}
\end{array}
$$

Logic Semantics:

The semantics of our logic is defined as follows:

$$
\begin{array}{lll}
e_1 e_2 \ldots e_n \models e & \text{iff} & e = e_n \\
\tau \models \varphi \ \wedge/\vee \ \psi & \text{iff} & \tau \models \varphi \text{ and/or } \tau \models \psi \\
e_1 e_2 \ldots e_n \models [t] & \text{iff} & (\exists 1 \leq i \leq n)\ (e_1 = \mathit{block}@t \text{ and} \\
 & & \quad (\forall i < j \leq n)\ e_j \neq \mathit{unblock}@t) \\
e_1 e_2 \ldots e_n \models \varphi \rightarrow \psi & \text{iff} & (\forall 1 \leq i \leq n)\ e_1 e_2 \ldots e_i \not\models \psi \text{ or} \\
 & & (\exists 1 \leq i \leq n)\ (e_1 e_2 \ldots e_i \models \psi \text{ and} \\
 & & \quad (\exists 1 \leq j \leq i)\ e_1 e_2 \ldots e_j \models \varphi)
\end{array}
$$

It is not hard to see that the two new operators $[t]$ and $\varphi \rightarrow \psi$ can be expressed in terms of PTLTL as

$$
\begin{array}{rcl}
[t] & \equiv & \neg \mathit{unblock}@t \ \mathcal{S} \ \mathit{block}@t \\
\varphi \rightarrow \psi & \equiv & \boxdot \neg \psi \ \vee \ \diamondsuit(\psi \wedge \diamondsuit \varphi)
\end{array}
$$

where $\mathcal{S}$ stands for "since" and $\boxdot$ for "always in the past".

UPCRC Illinois

Universal Parallel Computing Research Center

Manual Migration

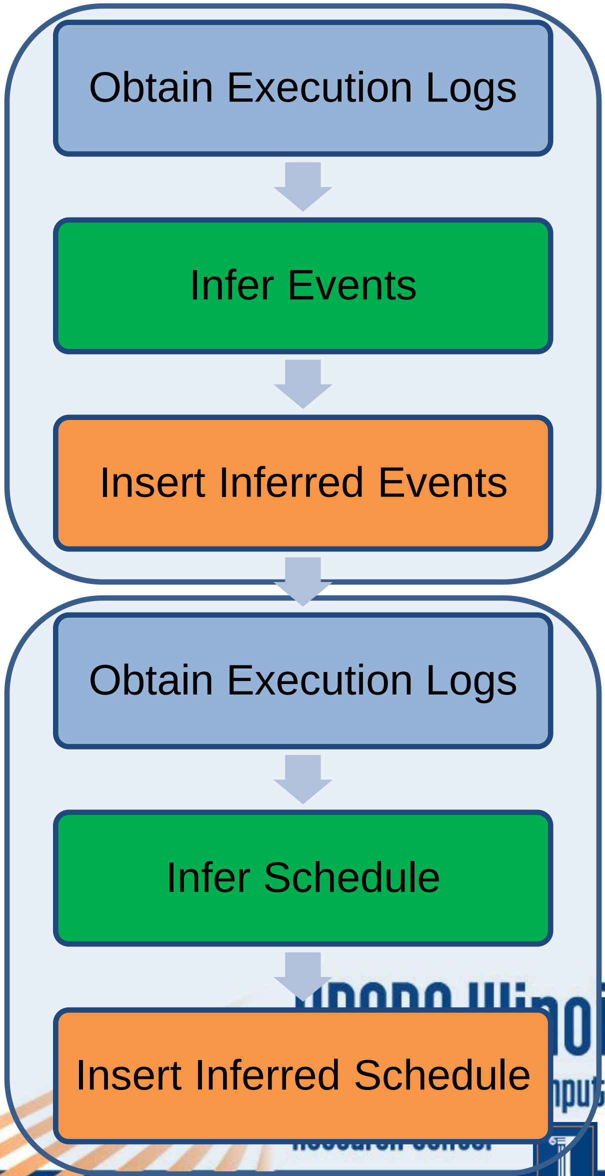
- We manually migrated over 200 sleep-based tests to IMUnit
- Migration typically involved the following steps:

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**

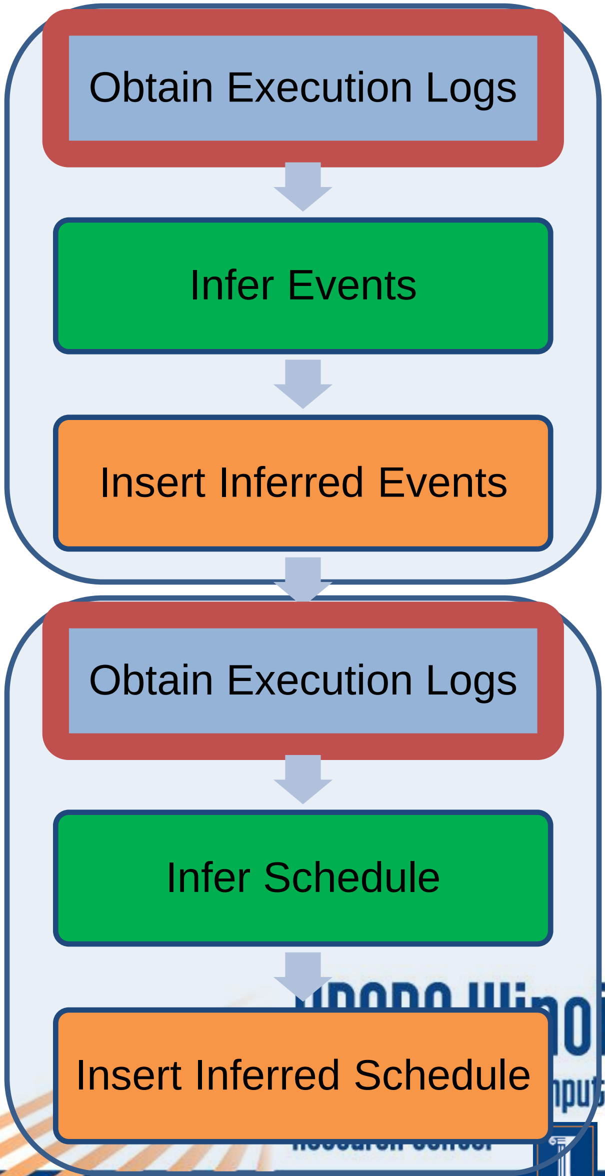
Automated Migration

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



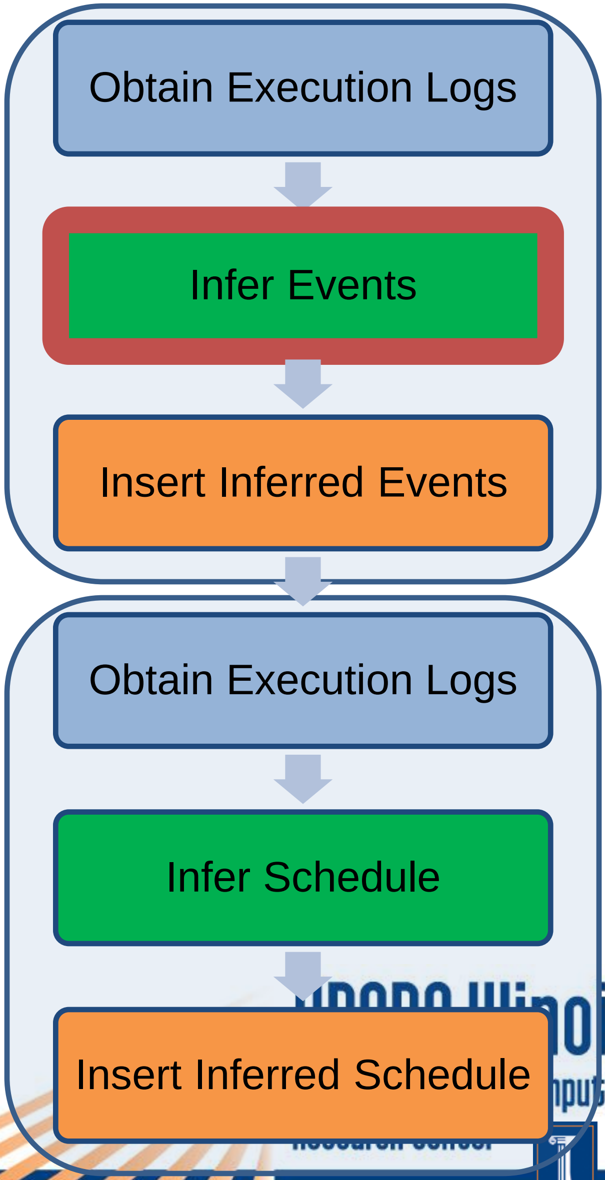
Logging

- 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)



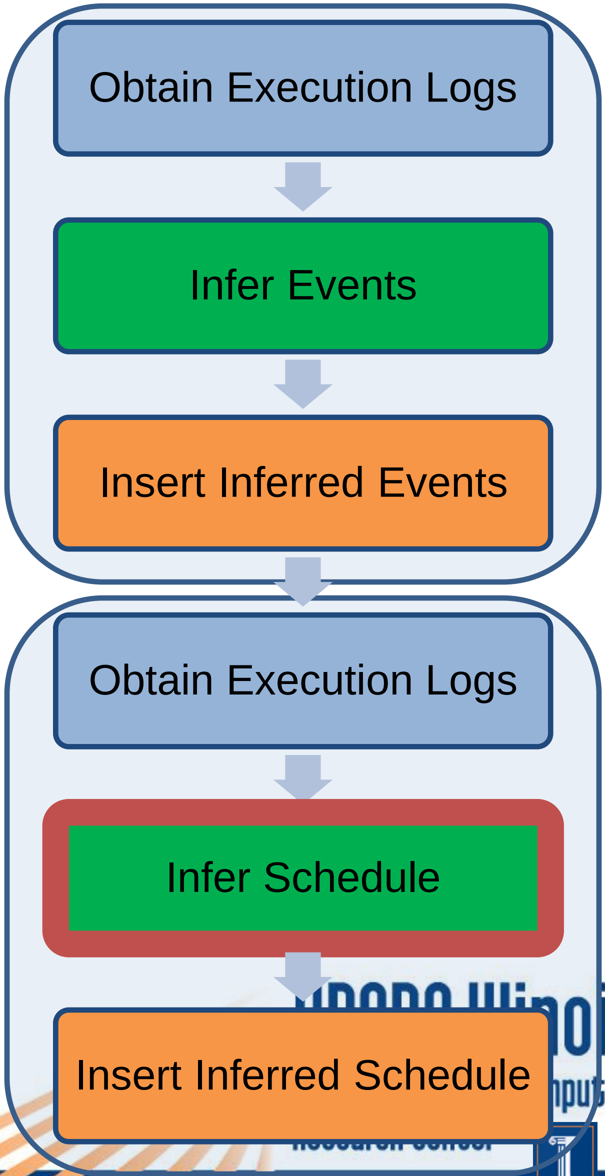
Events Inference

- 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



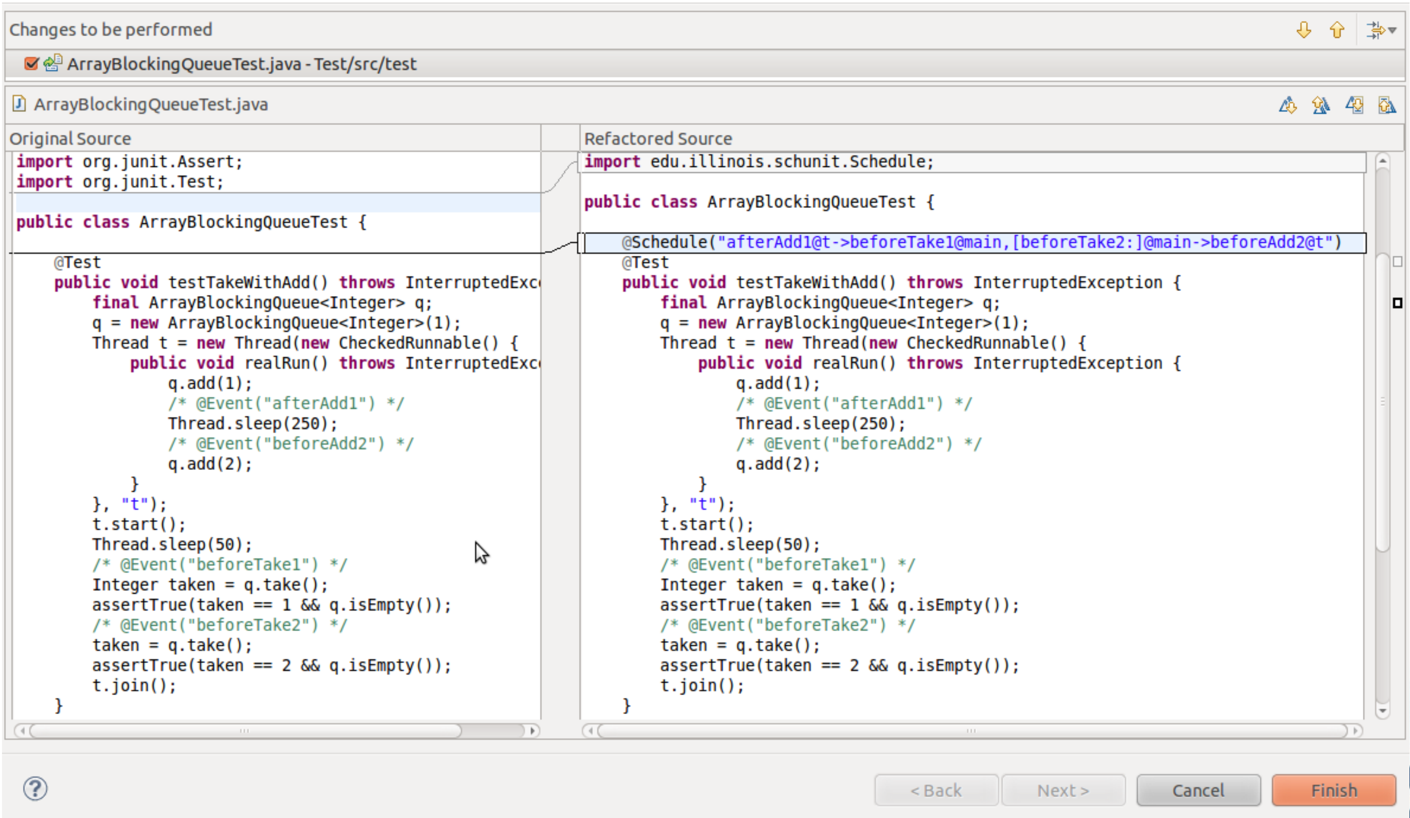
Schedule Inference

- 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:

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

afterAdd1->beforeTake1

[beforeTake2]->beforeAdd2

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

Subject	Description	Classes	Testcases
Collections	Apache Common Collections	2	18
JBoss-Cache	JBoss Cache Server	7	27
Lucene	Apache Lucene	1	2
Mina	Apache Mina	1	1
Pool	Apache Common Pool	2	2
Sysunit	Junit extension	5	9
TCK	JSR-166 TCK	18	139

Universal Parallel Computing Research Center

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

Subject	Inferring Events		Inferring Schedules	
	Precision	Recall	Precision	Recall
Collections	0.76	0.76	0.96	0.97
JBoss-Cache	0.81	0.84	0.87	0.96
Lucene	0.33	0.33	1.00	1.00
Mina	1.00	1.00	1.00	1.00
Pool	0.90	1.00	1.00	1.00
Sysunit	0.78	0.83	0.89	0.89
TCK	0.69	0.75	0.98	0.98
Arithmetic Mean	0.75	0.79	0.96	0.97

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

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

Universal Parallel Computing Research Center

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