Mutation Operators for Concurrent Java (J2SE 5.0)
(MUTATION 2006)

Jeremy S. Bradbury, James R. Cordy, Juergen Dingel
School of Computing, Queen’s University
Kingston, Ontario, Canada

SUPPORTED BY NSERC CRSMG
Motivation and Background

• In the next few years applications will need to be concurrent to fully exploit CPU throughput gains [Sut05]

• It is difficult to develop correct concurrent code
  
  “...humans are quickly overwhelmed by concurrency and find it much more difficult to reason about concurrent than sequential code. Even careful people miss possible interleavings...”

  - Herb Sutter & James Larus [SL05]

• Goal: use mutation to evaluate, compare and improve quality assurance techniques for concurrent Java.
Related Work

• Mutation with Java has been proposed in previous work
  – MuJava [MOK05]

• **Method** level mutation operators [MKO05]
  – statements (e.g., statement deletion)
  – operands and operators in expressions (e.g., operator insertion)

• **Class** level mutation operators [MKO02]
  – inheritance (e.g., super keyword deletion)
  – polymorphism (e.g., cast type change)
  – Java-specific features

• method and class level mutation operators do not **directly** mutate the source code in Java that handle concurrency.

---


Our Approach

• Need a more comprehensive set of operators
• Additional operators are needed to reflect the types of bugs that often occur in concurrent programs.
• We present a set of concurrent operators for Java (J2SE 5.0)
• We believe our new set of concurrency mutation operators used in conjunction with existing operators is more comprehensive
Java Concurrency (Prior to J2SE 5.0)

- Java concurrency is built around the notion of **multi-threaded** programs
  - sleep(), yield(), join() can affect the status of a thread
- Concurrency supported primarily through **synchronized** methods and blocks
- Synchronization blocks can be used in combination with **implicit monitor locks**
  - Monitor methods: wait(), notify(), notifyAll()
Java Concurrency (J2SE 5.0)

- **Lock**
  - Same semantics as the implicit monitor locks
- **Semaphore**
  - Maintains a set of permits that restrict the number of threads accessing a resource
- **Latch**
  - Allows threads from a set to wait until other threads complete a set of operations
- **Barrier**
  - A point at which threads from a set wait until all other threads reach the point
- **Exchanger**
  - Allows for the exchange of objects between two threads at a given synchronization point
Java Concurrency (J2SE 5.0)

- **Built-in Concurrent Data Structures**
  - collection types including ConcurrentHashMap and five different BlockingQueues.
- **Built-in Thread Pools**
  - FixedThreadPool and an unbounded CachedThreadPool.
- **Atomic Variables**
  - can be used in place of synchronization
  - Each atomic variable type contains new methods to support concurrency.
    - e.g., AtomicInteger contains addAndGet(), getAndSet(), etc.
Bug Pattern Taxonomy for Java Concurrency [FNU03]

- Patterns based on common mistakes programmers make when developing concurrent code
- Used to classify bugs in an existing concurrency benchmark maintained by IBM Haifa
  - 40 programs (57 to 17000 loc)
  - student created programs, tool developer programs, open source programs, and a commercial product
- Developed prior to J2SE 5.0
  - We added additional patterns that occur in new concurrency constructs

Bug Pattern Taxonomy for Java Concurrency

- Nonatomic operations assumed to be atomic
- Two-state access
- Wrong lock or no lock
- Double-checked lock
- The sleep() bug
- Losing a notify
- Other missing or nonexistent signals
- Notify instead of notify all
- A “blocking” critical section
- The orphaned thread
- Interference
- Deadlock (deadly embrace)
- Starvation
- Resource exhaustion
- Incorrect count initialization
Bug Pattern Taxonomy for Java Concurrency

• Nonatomic operations assumed to be atomic
• Wrong lock or no lock
• Notify instead of notify all
• A “blocking” critical section
• Deadlock (deadly embrace)
Mutation Operators for Concurrent Java

- Five categories of mutation operators for concurrent Java:
  1. modify parameters of concurrent methods
  2. modify the occurrence of concurrency method calls (removing, replacing, exchanging)
  3. Modify concurrency keywords (addition and removal)
  4. switch concurrent objects
  5. modify critical regions (shift, expand, shrink, split)
# Mutation Operators for Concurrent Java

The relationship between mutation operators for concurrency and the concurrency features provided by J2SE 5.0

<table>
<thead>
<tr>
<th>Java (J2SE 5.0) Concurrency Mutation Operator Categories</th>
<th>Threads</th>
<th>Synchronization methods</th>
<th>Synchronization statements</th>
<th>Synchronization with implicit monitor locks</th>
<th>Explicit locks</th>
<th>Semaphores</th>
<th>Barriers</th>
<th>Latches</th>
<th>Exchangers</th>
<th>Built-in concurrent data structures (e.g. queues)</th>
<th>Built-in thread pools</th>
<th>Atomic variables (e.g. Long/Integer)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Modify Parameters of Concurrent Methods</td>
<td>✓</td>
<td>–</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
</tr>
<tr>
<td>Modify the Occurrence of Concurrency Method Calls</td>
<td>✓</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>✓</td>
</tr>
<tr>
<td>Modify Keyword</td>
<td>–</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
</tr>
<tr>
<td>Switch Concurrent Objects</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Modify Concurrent Region</td>
<td>–</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
</tr>
</tbody>
</table>
Example Mutation: Modify Method Parameter

MSP – Modify Synchronized Block Parameter

//two lock objects created
private Object lock1 = new Object();
private Object lock2 = new Object();
....
//lock1 used in methodA
public void methodA() {
    synchronized (lock1) { ... }
}
...
Example Mutation: Modify Method Parameter
MSP – Modify Synchronized Block Parameter

//two lock objects created
private Object lock1 = new Object();
private Object lock2 = new Object();
....
//lock1 used in methodA
public void methodA() {
    synchronized (lock1) { ... }
}
...

//two lock objects created
private Object lock1 = new Object();
private Object lock2 = new Object();
....
//replace lock1 with lock2
public void methodA() {
    synchronized (lock2) { ... }
}
...
Example Mutation: Modify Method Call
RNA – Replace NotifyAll with Notify

//notifyAll is used to wake up all waiting
//threads

...notifyAll();...
Example Mutation: Modify Method Call
RNA – Replace NotifyAll with Notify

//notifyAll is used to wake up all waiting threads
...notifyAll();...

//notify will wake up only one waiting thread
...notify();...
Example Mutation: Modify Keyword
RFU – Remove Finally around Unlock

```java
...
private Lock lock1 = new ReentrantLock();
...
lock1.lock();
try {
    //critical region
    ...
} finally {
    lock1.unlock();
}
...
```
Example Mutation: Modify Keyword
RFU – Remove Finally around Unlock

```java
private Lock lock1
    = new ReentrantLock();

lock1.lock();
try {
    //critical region
    ...
} finally {
    lock1.unlock();
}
```

```java
private Lock lock1
    = new ReentrantLock();

lock1.lock();
try {
    //critical region
    ...
} //finally removed
lock1.unlock();
```

© J.S. Bradbury ● MUTATION 2006 ● 18
Example Mutation: Switch Concurrent Objects
EELO – Exchange Explicit Lock Objects

```java
private Lock lock1
    = new ReentrantLock();
private Lock lock2
    = new ReentrantLock();
...
lock1.lock();
...
lock2.lock();
...
finally {
    lock2.unlock();
}
...
finally {
    lock1.unlock();
}
...
```
Example Mutation: Switch Concurrent Objects
EELO – Exchange Explicit Lock Objects

```java
private Lock lock1
    = new ReentrantLock();
private Lock lock2
    = new ReentrantLock();

... 
lock1.lock();
...
lock2.lock();
...
finally {
    lock2.unlock();
}
...
finally {
    lock1.unlock();
}
...
```

```java
private Lock lock1
    = new ReentrantLock();
private Lock lock2
    = new ReentrantLock();

...
lock2.lock(); //should be lock1
...
lock1.lock(); //should be lock2
...
finally {
    lock2.unlock();
    lock1.unlock();
}
...
```
Example Mutation: Modify Critical Region
SKCR – Shrink Critical Region

```java
private Lock lock1
    = new ReentrantLock();
...
public void m1 () {
    <statement n1>
    lock1.lock();
    try {
        //critical region
        <statement c1>
        <statement c2>
        <statement c3>
    } finally {
        lock1.unlock();
    }
    <statement n2>
...
Example Mutation: Modify Critical Region

SKCR – Shrink Critical Region

```
private Lock lock1
    = new ReentrantLock();
...
public void m1 () {
    <statement n1>
    lock1.lock();
    try {
        //critical region
        <statement c1>
        <statement c2>
        <statement c3>
    } finally {
        lock1.unlock();
    }
    <statement n2>
...
# Mutation Operators vs. Bug Patterns

<table>
<thead>
<tr>
<th>Concurrency Bug Pattern</th>
<th>Mutation Operators</th>
</tr>
</thead>
<tbody>
<tr>
<td>Nonatomic operations assumed to be atomic bug pattern</td>
<td>RVK, EAN</td>
</tr>
<tr>
<td>Two-stage access bug pattern</td>
<td>SPCR</td>
</tr>
<tr>
<td>Wrong lock or no lock bug pattern</td>
<td>MSP, ESP, EELO, SHCR, SKCR, EXCR, RSB, RSK, ASTK, RSTK, RCXC, RXO</td>
</tr>
<tr>
<td>Double-checked locking bug pattern</td>
<td>–</td>
</tr>
<tr>
<td>The sleep() bug pattern</td>
<td>MXT, RJS, RTXC</td>
</tr>
<tr>
<td>Losing a notify bug pattern</td>
<td>RTXC, RCXC</td>
</tr>
<tr>
<td>Notify instead of notify all bug pattern</td>
<td>RNA</td>
</tr>
<tr>
<td>Other missing or nonexistent signals bug pattern</td>
<td>MXC, MBR, RCXC</td>
</tr>
<tr>
<td>A “blocking” critical section bug pattern</td>
<td>RFU, RCXC</td>
</tr>
<tr>
<td>The orphaned thread bug pattern</td>
<td>–</td>
</tr>
<tr>
<td>The interference bug pattern</td>
<td>MXT, RTXC, RCXC</td>
</tr>
<tr>
<td>The deadlock (deadly embrace) bug pattern</td>
<td>ESP, EXCR, EELO, RXO</td>
</tr>
<tr>
<td>Starvation bug pattern</td>
<td>MSF, ELPA</td>
</tr>
<tr>
<td>Resource exhaustion bug pattern</td>
<td>MXC</td>
</tr>
<tr>
<td>Incorrect count initialization bug pattern</td>
<td>M XC</td>
</tr>
</tbody>
</table>
Conclusions and Future Work

- Presented a set of 24 concurrency mutation operators
  - Can be used as both a comparative metric and coverage criteria
- Our operators are comprehensive and representative of bugs in an existing bug pattern taxonomy
- Concurrency operators are a complement not a replacement for existing operators
  - e.g., the concurrency operators can cause direct concurrency bugs while method and class operators can cause indirect concurrency bugs
- Currently we have implemented almost all of the operators.
- We are currently conducting empirical evaluations.
Mutation Operators for Concurrent Java (J2SE 5.0) (MUTATION 2006)

Jeremy S. Bradbury, James R. Cordy, Juergen Dingel
School of Computing, Queen’s University
Kingston, Ontario, Canada