Introduction

Why concurrent software testing?

The roadmap of future CPUs lead to multicore chips. The burden of utilizing multiple CPUs falls on the software developer.

Everyone will, at some point, write software for a multiple CPU system in their near future.

“Unit Testing Concurrent Software” by William Pugh & Nathaniel Ayewah, the developers behind MultithreadedTC.

By viewing & studying an existing framework, we can be more concrete and less abstract.

Learn some tools already available to us.

Good starting point.
Concurrency Testing is Different

- New types of defects exist such as deadlocks and race conditions.
- Concerns exist over how the system interleaves threads, not so much about input/output.
- Even single-threaded applications have a large number of paths of execution, the move to multi-threaded applications causes this to explode rapidly.
Concurrency Testing is Different

The demands of testing a multithreaded system call for new tools and frameworks.

JUnit is great, but we lack a way to control multiple threads such that we are then able to interleave threads in a particular fashion for the SUT.

Enter: MultithreadedTC
You will most likely use commonly available tools and frameworks while developing software.

JUnit, Cobertura, JavaNCSS, etc.

MultithreadedTC is the Java framework which allows the developer to write a specific sequence of interleaving threads to test the system for concurrency conditions.

Motivated by the philosophy that multithreaded applications should be built using small abstractions: semaphores, bounded buffers, latches, etc.

This means it’s possible to exercise/test all the possible combinations since the abstractions are small. The application logic and concurrency logic are separated.
MultithreadedTC Features

- Already developers are adding in features.
- Well documented.
- Easy to read and understand.
- Pure Java, no scripting language required (unlike ConAn).
- Eliminates “scaffolding code” needed to run multithreaded tests.
- No “.start()” or “.join()”, among others.
MultithreadedTC Documentation

Overview | Package | Class | Use | Tree | Deprecated | Index | Help
PREV CLASS | NEXT CLASS | SUMMARY: NESTED | FIELD | CONSTR | METHOD

edu.umd.cs.mtc

Class MultithreadedTestCase

djava.lang.Object
    \- junit.framework.Assert
    \- edu.umd.cs.mtc.MultithreadedTestCase

Direct Known Subclasses:
    MultithreadedTest

public abstract class MultithreadedTestCase
extends Assert

This is the base class for each test in the MultithreadedTC framework. To create a multithreaded test case, simply extend this class with "thread", that has no parameters and a void return type is a thread method. Each thread method will be run in a separate thread with \texttt{initialize()} and \texttt{finish()} methods you can override.

A single run of a multithreaded test case consists of:

1. Running the \texttt{initialize()} method
2. Running each thread method in a separate thread
3. Running the \texttt{finish()} method when all threads are done.

The method \texttt{TestFramework.runOnce(MultithreadedTestCase)} can be used to run a MultithreadedTestCase once. The method \texttt{TestFramework.runManyTimes(MultithreadedTestCase, int)} can be used to run a multithread test case multiple times (\texttt{int} different behaviors).
Both threads start at the same time. thread1() runs until thread2() sets the AtomicInteger to 2, then thread1() sets the ai to 3, and both threads join, then the final finish() method asserts that the ai is now 3.
MultithreadedTC is able to provide a nice platform for writing test cases for concurrent software, making the job of the software developer easier when test time comes.

This is due to a “metronome” the framework runs on a separate thread.

- The clock advances in “ticks” (from 0 to n).
- This allows the developer to correctly interleave threads.
- The metronome only advances to next tick when all threads are in a blocked state, and at least 1 thread is for a future tick.
Let us validate some properties of a bounded blocking buffer with a capacity of 1 element. We want to ensure that:

(a) The assertion `take = 42` occurs after the call to `put 17`
(b) The call to `put 17` blocks thread 1

**Solution 1:** Use `Thread.sleep()` to delay the first statement in thread 2. This introduces unnecessary timing dependence (test does not work well in a debugger or with an ill-timed garbage collector).

**Solution 2:** Use a latch to coordinate activities in both threads. This will not work because the call to `put 17` blocks thread 1 before the latch can be released.

**Solution 3:** Use MultithreadedTC!

```java
class BoundedBufferTest extends MultithreadedTestCase {
    BoundedBuffer buf;
    void initialize() { buf = new BoundedBuffer(1); }
    public void thread1() {
        buf.put(42);
        buf.put(17);
        assertTick(1);
    }
    public void thread2() {
        waitForTick(1);
        assertTrue(buf.take() == 42);
        assertTrue(buf.take() == 17);
    }
    void finish() { assertTrue(buf.isEmpty()); }
}
```

all tests extend base class
verify unblocking does not occur until tick 1
run simultaneously in different threads
JUnit Test
public void testBoundedBuffer() throws Throwable {
    TestFramework.runOnce(
        new BoundedBufferTest());
}
Authors evaluate the performance of MultithreadedTC by reproducing some test code used in the TCK for JSR 166.

<table>
<thead>
<tr>
<th>File</th>
<th>Rev.</th>
<th>Age</th>
<th>Author</th>
<th>Last log entry</th>
</tr>
</thead>
<tbody>
<tr>
<td>Parent Directory</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>AbstractExecutorServiceTest.java</td>
<td>1.21</td>
<td>9 days</td>
<td>jsr166</td>
<td>improve exception handling</td>
</tr>
<tr>
<td>AbstractQueueTest.java</td>
<td>1.3</td>
<td>9 days</td>
<td>jsr166</td>
<td>improve exception handling</td>
</tr>
<tr>
<td>AbstractQueuedLongSynchronizerTest.java</td>
<td>1.14</td>
<td>16 hours</td>
<td>jsr166</td>
<td>replace absolute waits with _DELAY_MS; 1000 =&gt; 1000L; short delay at</td>
</tr>
<tr>
<td>AbstractQueuedSynchronizerTest.java</td>
<td>1.32</td>
<td>16 hours</td>
<td>jsr166</td>
<td>replace absolute waits with _DELAY_MS; 1000 =&gt; 1000L; short delay at</td>
</tr>
<tr>
<td>ArrayBlockingQueueTest.java</td>
<td>1.26</td>
<td>8 days</td>
<td>jsr166</td>
<td>use autoboxing judiciously for readability</td>
</tr>
<tr>
<td>ArrayDequeTest.java</td>
<td>1.12</td>
<td>8 days</td>
<td>jsr166</td>
<td>use autoboxing judiciously for readability</td>
</tr>
<tr>
<td>AtomicBooleanTest.java</td>
<td>1.15</td>
<td>9 days</td>
<td>jsr166</td>
<td>untabify</td>
</tr>
<tr>
<td>AtomicIntegerArrayTest.java</td>
<td>1.15</td>
<td>13 days</td>
<td>jsr166</td>
<td>Runnable =&gt; CheckedRunnable</td>
</tr>
<tr>
<td>AtomicIntegerFieldUpdaterTest.java</td>
<td>1.16</td>
<td>9 days</td>
<td>jsr166</td>
<td>improve exception handling</td>
</tr>
<tr>
<td>AtomicIntegerTest.java</td>
<td>1.18</td>
<td>8 days</td>
<td>jsr166</td>
<td>More thorough testing of values</td>
</tr>
<tr>
<td>AtomicLongArrayTest.java</td>
<td>1.14</td>
<td>13 days</td>
<td>jsr166</td>
<td>Runnable =&gt; CheckedRunnable</td>
</tr>
<tr>
<td>AtomicLongFieldUpdaterTest.java</td>
<td>1.16</td>
<td>9 days</td>
<td>jsr166</td>
<td>improve exception handling</td>
</tr>
<tr>
<td>AtomicLongTest.java</td>
<td>1.17</td>
<td>8 days</td>
<td>jsr166</td>
<td>More thorough testing of values</td>
</tr>
<tr>
<td>AtomicMemberReferenceTest.java</td>
<td>1.12</td>
<td>9 days</td>
<td>jsr166</td>
<td>untabify</td>
</tr>
</tbody>
</table>
Results of Evaluation

Of those tests in the TCK, the authors looked at 258 tests which attempted to use a specific interleaving of threads in 33 classes, and implemented those tests using MultithreadedTC.

<table>
<thead>
<tr>
<th>Measure</th>
<th>TCK</th>
<th>MTC</th>
</tr>
</thead>
<tbody>
<tr>
<td>Lines of Code</td>
<td>8003</td>
<td>7070</td>
</tr>
<tr>
<td>Bytecode Size</td>
<td>1017K</td>
<td>980K</td>
</tr>
<tr>
<td>*Local variables per method</td>
<td>1.12</td>
<td>0.12</td>
</tr>
<tr>
<td>*Av. anon. inner classes/method</td>
<td>0.38</td>
<td>0.01</td>
</tr>
</tbody>
</table>

* Metrics measured by the software quality tool Swat4j [4]
MultithreadedTC vs ConAn
Concurrency Analyzer (ConAn)

- Testing Tool.
- Developed by Strooper, Duke, Wildman, Goldson and Long at the University of Queensland, Australia.
- Based on Roast.
- Aims for short test cases.
- Script-based.
- Uses internal clock for synchronization.
- Generates Java test driver from test script.
Test Driver Generation

ConAn
Test Script

Roast

ConAn

Test Driver
Example

```c
#ticktime 200
#monitor m WriterPreferenceReadWriteLock
...
\ldots
#begin
  #test C1 C13
  #tick
  #thread <t1>
  #excMonitor m.readLock().attempt(1000); #end
  #valueCheck time() # 1 #end
  #end
#end
#tick
#thread <t1>
#excMonitor m.readLock().release(); #end
#valueCheck time() # 2 #end
#end
#end
```

Figure 4: The script for a ConAn test to validate a Writer Preference Read Write Lock
## Comparison

<table>
<thead>
<tr>
<th>ConAn</th>
<th>MultithreadedTC</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ticks at regular intervals</td>
<td>Advances when all threads are blocked</td>
</tr>
<tr>
<td>Organization by ticks</td>
<td>Organization by threads</td>
</tr>
<tr>
<td>Deterministic</td>
<td>Deterministic &amp; Indeterministic</td>
</tr>
<tr>
<td>Custom Syntax / Scripting Language</td>
<td>Pure Java</td>
</tr>
</tbody>
</table>
Comparison

Line count comparison between MultithreadedTC and ConAn for tests on the WriterPreferenceRead_WriteLock Java Class.

<table>
<thead>
<tr>
<th>Test Suite</th>
<th>MTC</th>
<th>ConAn</th>
<th>ConAn Driver</th>
</tr>
</thead>
<tbody>
<tr>
<td>Basic Tests</td>
<td>274</td>
<td>829</td>
<td>2192</td>
</tr>
<tr>
<td>Tests with Interrupts</td>
<td>456</td>
<td>1386</td>
<td>3535</td>
</tr>
<tr>
<td>Tests with Timeouts</td>
<td>389</td>
<td>585</td>
<td>1629</td>
</tr>
<tr>
<td>Total Line Count</td>
<td>1119</td>
<td>2800</td>
<td>7356</td>
</tr>
</tbody>
</table>
Other Testing Tools

JUnit, TestNG, GroboUtils, ConTest
JUnit

junit.extensions.ActiveTestSuite

- Build a suite of tests that run concurrently
- Run each test in a separate thread
- Suite does not finish until all test threads are complete
- Can be used with RepeatedTest to uncover threading problems

```java
public static Test suite() {
    TestSuite suite = new ActiveTestSuite();
    suite.addTest(new TestGame("testCreateFighter"));
    suite.addTest(new TestGame("testGameInitialState"));
    suite.addTest(new TestGame("testSameFighters"));
    return suite;
}
```
TestNG

- Inspired by JUnit
- Tests are run in parallel
  - Parallel parameter must be set
- Thread-count
  - Specifies size of thread pool

```java
@Test(threadPoolSize = 3, invocationCount = 10, timeOut = 200)
public void shouldHandleRequestSwiftly() {
    // make request
}
```
Extension of JUnit.

Thread is specified by extending TestRunnable class and implementing runTest() method.

Classes are provided to run multiple instances of the thread simultaneously.

Monitors execution of threads to look for inconsistent states.
ConTest

Developed by IBM.

Records and replays interleavings that lead to faults.

Uses sleep() and yield() to test different interleavings each time a test is run.
Conclusion

- MultithreadedTC
  - Short test cases.
  - JUnit compatible.
  - Involves very little overhead.


JSR 166 Interest Site (maintained by Doug Lea): http://gee.cs.oswego.edu/dl/concurrency-interest/