Unit Testing Concurrent Software

The MultithreadedTC Framework

Ricardo Miron and Jack Weaver for Comp 587

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 interleave threads in a particular fashion for the SUT.
- Enter: MultithreadedTC

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

- Open source: http://code.google.com/p/multithreadedtc/
  - 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

edu.umd.cs.mtc

Class MultithreadedTestCase

java.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 cl
with "thread", that has no parameters and a void return type is a thread method. Each thread method will be run in a seperate th
`initialize()` and `finish()` methods you can override.

A single run of a multithreaded test case consists of:

1. Running the `initialize()` method
2. Running each thread method in a seperate thread
3. Running the `finish()` method when all threads are done.

The method `TestFramework.runOnce(MultithreadedTestCase)` can be used to run a MultithreadedTestCase once. The m
`TestFramework.runManyTimes(MultithreadedTestCase, int)` can be used to run a multithread test case multiple times (t
different behaviors).

MultithreadedTC Code Example

MultithreadedTC Version

```
class MTCCompareAndSet extends MultithreadedTest {
    AtomicInteger ai;
    @Override public void initialize() {
        ai = new AtomicInteger(1);
    }

    public void thread1() {
        while(!ai.compareAndSet(2, 3)) Thread.yield();
    }

    public void thread2() {
        assertTrue(ai.compareAndSet(1, 2));
    }

    @Override public void finish() {
        assertEquals(ai.get(), 3);
    }
}
```

Stays in loop until condition is met by 2nd thread

Plain Version

```
public void testCompareAndSet() throws InterruptedException {
    final AtomicInteger ai = new AtomicInteger(1);
    Thread t = new Thread(new Runnable() {
        public void run() {
            while(!ai.compareAndSet(2, 3))
                Thread.yield();
        }
    });

    t.start();
    assertTrue(ai.compareAndSet(1, 2));
    t.join(2500);
    assertFalse(t.isAlive());
    assertEquals(ai.get(), 3);
}
```

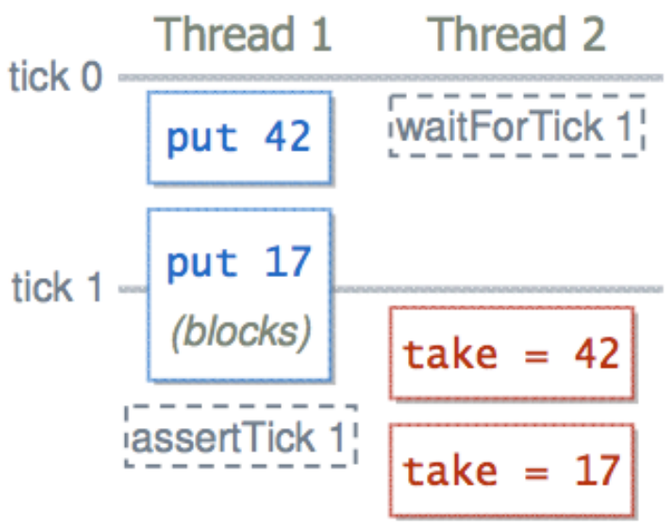
Both threads start at the same time. thread1() runs until thread2() sets the AtomicInteger to 2, then thread1() sets the ai to 3, and both threads join, then the final finish() method asserts that the ai is now 3.

MultithreadedTC Clock

- 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.

MultithreadedTC Example

EXAMPLE



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!

```
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

waits until all threads are blocked

JUnit Test

```
public void testBoundedBuffer()
 throws Throwable {
    TestFramework.runOnce(
        new BoundedBufferTest()
    );
}
```

Framework Evaluation

- Authors evaluate the performance of MultithreadedTC by reproducing some test code used in the TCK for JSR 166.

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

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 1: Overall comparison of TCK tests and MTC (MultithreadedTC) implementation

Measure	TCK	MTC
Lines of Code	8003	7070
Bytecode Size	1017K	980K
*Local variables per method	1.12	0.12
*Av. anon. inner classes/method	0.38	0.01

** Metrics measured by the software quality tool Swat4j [4]*

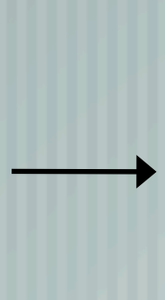
MultithreadedTC vs ConAn

Concurrency Analyzer (ConAn)

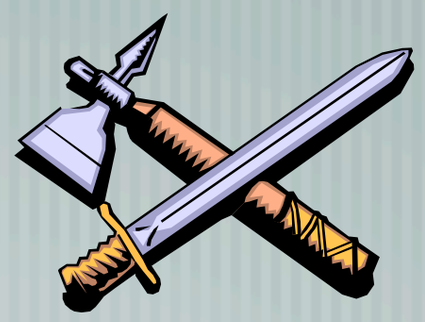
- Testing Tool.
- Developed by Strooper, Duke, Wildman, Goldson and Long at the University of Queensland, Australia.
- Based on Roast.
- Aims for short test cases.
- Script-based.
- Uses internal clock for synchronization.
- Generates Java test driver from test script.

Test Driver Generation

ConAn
Test Script



Roast



ConAn

Test Driver

Example

```
#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
#end
```

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

Comparison

ConAn	MultithreadedTC
Ticks at regular intervals	Advances when all threads are blocked
Organization by ticks	Organization by threads
Deterministic	Deterministic & Indeterministic
Custom Syntax / Scripting Language	Pure Java

Comparison

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

Test Suite	MTC	ConAn	ConAn Driver
Basic Tests	274	829	2192
Tests with Interrupts	456	1386	3535
Tests with Timeouts	389	585	1629
Total Line Count	1119	2800	7356

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

```
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

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

GroboUtils

- 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.

References & More Information

- "Viewpoint: Face the inevitable, embrace parallelism", ACM: http://portal.acm.org/citation.cfm?id=1562164.1562179&coll=GUIDE&dl=GUIDE&CFID=66231572&CFTOKEN=60199555

- MultithreadedTC: http://www.cs.umd.edu/projects/PL/multithreadedtc/index.html

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

- WriterPreferenceReadWriteLock.java, by Doug Lea : http://gee.cs.oswego.edu/dl/classes/EDU/oswego/cs/dl/util/concurrent/WriterPreferenceReadWriteLock.java