Unit Testing Concurrent Software

William Pugh
Dept. of Computer Science
Univ. of Maryland
College Park, MD
pugh@cs.umd.edu

Nathaniel Ayewah
Dept. of Computer Science
Univ. of Maryland
College Park, MD
ayewah@cs.umd.edu

ABSTRACT

There are many difficulties associated with developing correct multithreaded software, and many of the activities that are simple for single threaded software are exceptionally hard for multithreaded software. One such example is constructing unit tests involving multiple threads. Given, for example, a blocking queue implementation, writing a test case to show that it blocks and unblocks appropriately using existing testing frameworks is exceptionally hard. In this paper, we describe the MultithreadedTC framework which allows the construction of deterministic and repeatable unit tests for concurrent abstractions. This framework is not designed to test for synchronization errors that lead to rare probabilistic faults under concurrent stress. Rather, this framework allows us to demonstrate that code does provide specific concurrent functionality (e.g., a thread attempting to acquire a lock is blocked if another thread has the lock).

We describe the framework and provide empirical comparisons against hand-coded tests designed for Sun's Java concurrency utilities library and against previous frameworks that addressed this same issue. The source code for this framework is available under an open source license.

Categories and Subject Descriptors

D.2.5 [**Software Engineering**]: Testing and Debugging—*testing tools*

General Terms

Experimentation, Reliability, Verification

Keywords

Java, testing framework, concurrent abstraction, JUnit test cases, MultithreadedTC

1. INTRODUCTION

Concurrent applications are often hard to write and even harder to verify. The application developer has to adhere to various locking protocols, avoid race conditions, and coordinate multiple threads. Many developers manage this complexity by separating the concurrent logic from the business logic, limiting concurrency to small abstractions like latches, semaphores and bounded buffers. These abstractions can then be tested independently, adequately validating the correctness of most of the concurrent aspects of the application.

One strategy for testing concurrent components is to write large test cases that contain many possible interleavings and run these test cases many times hopefully inducing some rare but faulty interleavings. Many concurrent test frameworks are based on this paradigm and provide facilities to increase the diversity of interleavings [5, 9]. But this strategy may miss some interleavings that lead to failures, and do not yield consistent results.

A different strategy is to test specific interleavings separately. But some critical interleavings are hard to exercise because of the presence of blocking and/or timing. This paper describes MultithreadedTC, a framework that allows a test designer to exercise a specific interleaving of threads in an application. It features a metronome (or clock) that allows test designers to coordinate threads even in the presence of blocking and timing issues. The clock advances when all threads are blocked; test designers can delay operations within a thread until the clock has reached a desired tick. MultithreadedTC also features a concise syntax for specifying threads and eliminates much of the scaffolding code required to make multithreaded tests work well with JUnit. It can also detect deadlocks and livelocks.

2. A SIMPLE EXAMPLE

Figure 1 shows four operations on a bounded buffer, distributed over two threads. A bounded buffer allows users to `take` elements that have been `put` into the container, in the order in which they were put. It has a fixed capacity and both `put` and `take` cause the calling thread to *block* if the container is full or empty respectively.

In this example, the call to `put 17` should block thread 1 until the assertion `take = 42` in thread 2 frees up space in the bounded buffer. But how does the test designer guarantee that thread 1 blocks before thread 2's assertion, and does not unblock until after the assertion?

One option is to put thread 2 to sleep for a fixed time period long enough to "guarantee" that thread 1 has blocked. But this introduces unnecessary timing dependence — the resulting code does not play well in a debugger, for instance — and the presence of multiple timing dependent units would make a large example harder to understand.

ASE'07, November 4–9, 2007, Atlanta, Georgia, USA.

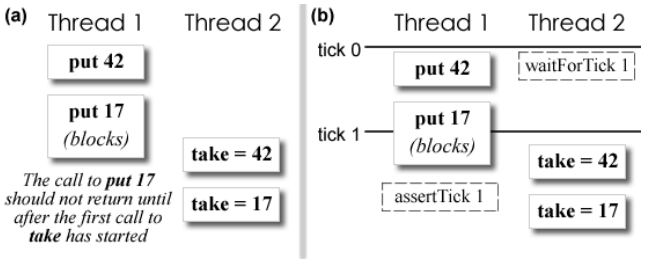

Figure 1: (a) Operations in multiple threads on a bounded buffer of capacity 1 must occur in a particular order, (b) MultithreadedTC guarantees the correct order.

Another approach is to make thread 2 wait on a *latch* that is released at the appropriate time. But this will not work here because the only other thread available to release the latch (thread 1) is blocked.

MultithreadedTC's solution is to superimpose an external clock on both threads. The clock runs in a separate (daemon) thread which periodically checks the status of all test threads. If all the threads are blocked, and at least one thread is waiting for a tick, the clock advances to the next desired tick. In Figure 1, thread 2 blocks immediately, and thread 1 blocks on the call to `put 17`. At this point, all threads are blocked with thread 2 waiting for tick 1. So the clock thread advances the clock to tick 1 and thread 2 is free to go. Thread 2 then takes from the buffer with the assertion `take = 42`, and this releases Thread 1. Hence with the test, we are confident that (1) thread 1 blocked (because the clock advanced) and (2) thread 1 was not released until after thread 2's first assertion (because of the final assertion in thread 1).

MultithreadedTC is partly inspired by ConAn [11, 12, 13], a script-based concurrency testing framework which also uses a clock to coordinate the activities of multiple threads. We provide a comparison of ConAn and MultithreadedTC in Section 4.2.

3. IMPLEMENTATION DETAILS

Figure 2 shows a Java implementation of the test in Figure 1 and illustrates some of the basic components of a MultithreadTC test case. Each test is a subclass of MultithreadedTestCase. Threads are specified using *thread* methods which return void, have no arguments and have names prefixed with the word "thread". Each test may also override the `initialize()` or `finish()` methods provided by the base class. When a test is run, `initialize()` is invoked first, then all thread methods are invoked simultaneously in separate threads, and finally when all the thread methods have completed, `finish()` is invoked. This is analogous to the organization of setup, test and teardown methods in JUnit [2].

Each MultithreadedTestCase maintains a clock that starts at time 0. The clock advances when all threads are blocked and at least one thread is waiting for a tick. The method `waitForTick(i)` blocks the calling thread until the clock reaches tick `i`. A test designer can also temporarily prevent the clock from advancing when all threads are blocked by using `freezeClock()`.

```
class BoundedBufferTest extends MultithreadedTestCase {
    ArrayBlockingQueue<Integer> buf;
    @Override public void initialize() {
         buf = new ArrayBlockingQueue<Integer>(1);
    }
    public void thread1() throws InterruptedException {
         buf.put(42);
         buf.put(17);
         assertTick(1);
    }
    public void thread2() throws InterruptedException {
         waitForTick(1);
         assertTrue(buf.take() == 42);
         assertTrue(buf.take() == 17);
    }
    @Override public void finish() {
         assertTrue(buf.isEmpty());
    }
}

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

Figure 2: Java version of bounded buffer test.

The test sequence is run by invoking one of the run methods in TestFramework. (This can be done in a JUnit test, as in Figure 2.) TestFramework is also responsible for regulating the clock. It creates a thread that periodically checks the status of the clock and test threads and decides when to release threads that are waiting for a tick. It can also detect deadlock: when all threads are blocked and none of them is waiting for a tick or a timeout. More details on creating and running tests are available on the project website at [3].

4. TEST FRAMEWORK EVALUATION

To evaluate MultithreadedTC, we performed some qualitative and quantitative comparisons with other test frameworks, focusing on how easy it is to write and understand test cases, and how expressive each test framework is. Section 4.1 compares MultithreadedTC with TCK tests created to validate Java's concurrency utilities library [1]. We take this test suite to represent a typical JUnit-based implementation of multithreaded tests. Section 4.2 compares MultithreadedTC with ConAn.

4.1 JSR 166 TCK comparison

JSR (Java Specification Request) 166 is a set of proposals for adding a concurrency utilities library to Java [1]. This includes components like locks, latches, thread locals and bounded buffers. Its TCK (Technology Compatibility Kit) includes a suite of JUnit tests that validate the correctness of the library's implementation against the specification provided in the JSR. We examined 258 of these tests which attempt to exercise specific interleavings in 33 classes, and implemented them using MultithreadedTC. These tests do not measure performance, nor do they look for rare errors that may occur when a concurrent abstraction is under stress.

In general MultithreadedTC allows the test designer to use simpler constructs that require fewer lines of code. Our evaluation also demonstrates that MultithreadedTC can express all the strategies used in the TCK tests and is often more precise and expressive, especially for tests that do not require a timing dependency. Table 1 gives an overview of the comparison, including the observation that MultithreadedTC requires fewer local variables and anonymous inner

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]*

```
threadFailed = false;
...
Thread t = new Thread(new Runnable() {
   public void run() {
      try {
         // statement(s) that may cause exception
      } catch(InterruptedException ie) {
         threadFailed = true;
         fail("Unexpected exception");
      }
   }
});
t.start();
...
t.join();
assertFalse(threadFailed);
```

Figure 3: Handling threads in JUnit tests

classes, in part because it does not make the test designer construct threads manually. The TCK tests generally use anonymous inner classes as illustrated in Figure 3. This syntax is quite verbose, especially since all the useful functionality is in the method `run()`.

Another source of verbosity in Figure 3 is the scaffolding required to handle exceptions. While exceptions in MultithreadedTC cause the test to fail immediately, exceptions in a JUnit test thread will kill the thread but will not fail the test. To force failure, the TCK tests use additional `try-catch` blocks to catch thread exceptions and set a flag (such as the `threadFailed` flag in Figure 3). MultithreadedTC also eliminates the last *join* statement in Figure 3 because it checks to make sure all threads complete and uses `finish()` to execute code that must run after all threads complete.

Some of the differences we have just described are quantified in Table 2. This table includes simple counts of the number of TCK tests that use the constructs described above (anonymous inner classes, try-catch blocks and joins) in a way that is eliminated by the corresponding MultithreadedTC test. It also counts the total number of times these constructs are removed.

4.2 Comparison to ConAn

Table 2: The number of TCK tests with constructs that were removed in the MultithreadedTC version and the total number of constructs removed.

Construct Removed	Tests	Removed
Anonymous inner classes	216	257
Thread's `join()` method	193	239
`try-catch` blocks	104	106
Thread's `sleep()` method	198	313

```
#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

ConAn (Concurrency Analyzer) is a script-based test framework that, like MultithreadedTC, uses a clock to synchronize the actions in multiple threads [12]. ConAn also aims to make it possible to write succinct test cases by introducing a script-based syntax illustrated in Figure 4. To run the test, the script is parsed and translated into a Java test driver containing methods which interact with some predefined ConAn library classes and implement the tests specified in the script.

Each test is broken into *tick* blocks that contain one or more *thread* blocks (which may have optional labels). A thread block may span multiple tick blocks by using the same label. Thread blocks contain valid Java code or use custom script tags to perform common operations like handling exceptions (excMonitor) and asserting values and equalities (valueCheck). Any blocking statements in a given tick may unblock at a later tick, which is confirmed by checking the time.

The major difference between the two frameworks is that ConAn relies on a timer-based clock which ticks at regular intervals, while MultithreadedTC advances the clock when all threads are blocked. ConAn's strategy introduces a timing dependence even in tests that do not involve any timing like the bounded buffer example in Figure 1.

The two frameworks also use different paradigms to organize tests. ConAn organizes tests by ticks, while MultithreadedTC organizes tests by threads. The two paradigms are difficult to compare quantitatively as each involves tradeoffs for the test designer. MultithreadedTC allows designers to inadvertently introduce indeterminism into tests by placing a waitForTick at the wrong place, while ConAn forces designers to deterministically put code segments into the tick in which they are to run. On the other hand ConAn's paradigm can be confusing when writing tests because the code in one thread is spread out spatially into many ticks, while MultithreadedTC places all the code for a thread into one method, consistent with Java's paradigm. Also, ConAn's organization hardcodes the content of ticks and does not allow for "relative" time which is useful if blocking occurs in a loop.

Another significant difference between the two frameworks is that ConAn relies on a custom script-based syntax, while MultithreadedTC is pure Java and borrows metaphors from JUnit. This means that test designers using ConAn do not have access to many of the powerful features provided in

Table 3: Line count comparisons between MultithreadedTC and ConAn for tests on the Writer Preference Read Write Lock

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

modern integrated development environments (IDEs) such as refactoring, syntax highlighting, and code completion.

ConAn was partially evaluated using tests written to validate an implementation of the WriterPreferenceReadWriteLock. We implemented MultithreadedTC versions of these tests and compared the number lines of code of the two implementations. This comparison, shown in Table 3 indicates that the MultithreadedTC tests are more succinct than the ConAn scripts and significantly more succinct than ConAn's generated Java drivers.

5. RELATED WORK

Many java test frameworks provide basic facilities that allow test designers to run unit tests with multiple threads but do not remove the resulting nondeterminism. JUnit [2] provides an *ActiveTestSuite* extension which runs all the tests in the suite simultaneously in different threads and waits for the threads to complete. In TestNG [6], tests are run in parallel if the *parallel* parameter is set. An additional parameter, *thread-count*, is used to specify the size of the thread pool. Another framework, GroboUtils [5] extends the JUnit framework to provide support for writing thread safe multithreaded tests without having to write much thread handling code. A thread is specified by extending a provided TestRunnable class and implementing the `runTest()` method. Other classes are provided to run multiple instances of this thread simultaneously, enforce a time limit (after which all threads are killed) and regularly monitor the execution of the threads to look for inconsistent states.

MultithreadedTC's effort to control the synchronization of multiple threads is partly inspired by ConAn [11, 12, 13]. ConAn extends a technique for testing concurrent monitors introduced by Brinch Hansen [10]. Both ConAn and Hansen's method use a clock that is incremented at regular time intervals and used to synchronize thread operations. But ConAn organizes tests into tick blocks while Hansen's method organizes tests into threads and uses await statements (which is the paradigm followed by MultithreadedTC).

Other frameworks and approaches have focused on allowing tests to run nondeterministically, recording the interleavings that fail, and deterministically replaying them by transforming the original program. Carver and Tai [7] use a language-based approach (in Ada) that transforms the original program under test into an equivalent program that uses semaphores and monitors to control synchronization. ConTest [9] is a Java testing framework that uses a source level version of the deterministic replay algorithm introduced in DejaVu [8] to record and replay specific interleavings that lead to faults. Its algorithm also modifies the original program to add synchronization.

ConTest and some other frameworks like GroboUtils also provide facilities to run the tests many times in hopes of finding rare failing interleavings. ConTest relies on source level seeding using standard Java Thread methods like `sleep()` and `yield()` to increase the chance that a different interleaving is used each time the test is run. Another approach would be to take control of the thread scheduler, but this would require modifying the Java virtual machine, and make the final solution less portable and platform independent.

6. CONCLUSIONS

Concurrency in applications is receiving new emphasis as chip manufacturers develop multi-core processors. Software engineers need more effective approaches of creating fault free programs that exploit the level of concurrency these processors can offer. MultithreadedTC provides a Java based framework for writing tests that exercise specific interleavings of concurrent programs. It was designed for testing small concurrent abstractions in which all possible interleavings can be enumerated and tested separately. The framework is robust in the face of blocking and timing issues. The resulting tests are succinct, JUnit compatible, and involve very little overhead.

The source code and documentation for MultithreadedTC are available under an open source license at [3].

7. REFERENCES

[1] Jsr 166: Concurrency utilities. http://www.jcp.org/en/jsr/detail?id=166, 2004.

[2] Junit testing framework. http://www.junit.org, 2007.

[3] Multithreadedtc. http://code.google.com/p/multithreadedtc/, 2007.

[4] Swat4j. http://www.codeswat.com, 2007.

[5] M. Albrecht. Using multi-threaded tests. http://groboutils.sourceforge.net/testing-junit/using_mtt.html, September 2004.

[6] C. Beust and A. Popescu. Testng: Testing, the next generation. http://www.testng.org, 2007.

[7] R. H. Carver and K.-C. Tai. Replay and testing for concurrent programs. *IEEE Softw.*, 8(2):66–74, 1991.

[8] J.-D. Choi and H. Srinivasan. Deterministic replay of java multithreaded applications. In *SPDT '98: Proceedings of the SIGMETRICS symposium on Parallel and distributed tools*, pages 48–59, New York, NY, USA, 1998. ACM Press.

[9] O. Edelstein, E. Farchi, Y. Nir, G. Ratsaby, and S. Ur. Multithreaded java program test generation. *IBM Systems Journal*, 41(1):111–125, 2002.

[10] P. B. Hansen. Reproducible testing of monitor. *Softw., Pract. Exper.*, 8(6):721–729, 1978.

[11] B. Long. *Testing Concurrent Java Components*. PhD thesis, The University of Queensland, July 2005.

[12] B. Long, D. Hoffman, and P. Strooper. Tool support for testing concurrent java components. *IEEE Transactions on Software Engineering*, 29(6), 2003.

[13] L. Wildman, B. Long, and P. A. Strooper. Testing java interrupts and timed waits. In *APSEC*, pages 438–447, 2004.