ConcJUnit: Unit Testing for Concurrent Programs

Mathias Ricken
Dept. of Computer Science
Rice University
Houston, TX 77005, USA
+1-713-348-3836
mgricken@rice.edu

Robert Cartwright
Dept. of Computer Science
Rice University
Houston, TX 77005, USA
+1-713-348-6042
cork@rice.edu

ABSTRACT
In test-driven development, tests are written for each program unit before the code is written, ensuring that the code has a comprehensive unit testing harness. Unfortunately, unit testing is much less effective for concurrent programs than for conventional sequential programs, partly because extant unit testing frameworks provide little help in addressing the challenges of testing concurrent code. In this paper, we present ConcJUnit, an extension of the popular unit testing framework JUnit that simplifies the task of writing tests for concurrent programs by handling uncaught exceptions and failed assertions in all threads, and by detecting child threads that were not forced to terminate before the main thread ends.

Categories and Subject Descriptors
D.1.3 [**Programming Techniques**]: Concurrent Programming; D.2.5 [**Software Engineering**]: Testing and Debugging – *abstract data types, polymorphism, control structures.*

General Terms
Reliability, Languages.

Keywords
Java, JUnit, unit testing, concurrent programming.

1. INTRODUCTION
Incremental, test-driven development is the most distinctive feature of agile approaches to software development such as Extreme Programming [1]. Tests are written for a unit of code before the code itself is written, and all tests must succeed before a new revision can be committed to the code base, facilitating the early detection and repair of program bugs.

Unfortunately, unit testing is much less effective for programs with multiple threads of control because program execution becomes non-deterministic, subject to variations in thread scheduling. Extant unit testing frameworks provide little support for testing concurrent code. In fact, they (perhaps unwittingly) facilitate the writing of bad unit tests. In JUnit [2] and TestNG [3], concurrent unit tests that should fail often report success for a variety of reasons: They silently ignore uncaught exceptions and failed assertions that occur in threads other than the main thread, and they do not provide any warnings when spawned child threads fail to terminate before the test is declared a success.

PPPJ '09, August 27-28, 2009, Calgary, Alberta, Canada.

Contributions Our contributions are as follows.

We present ConcJUnit, a unit testing framework for Java that revises and extends JUnit.

- Uncaught exceptions and failed assertions are detected in all threads, not just in the test's main thread, and cause the unit test to fail (Section 2).
- Child threads are required to end before the test result is determined. ConcJUnit emits a warning if a child thread outlives the test or ended in time but was not forced to do so (Section 3).

Our implementation is backward-compatible to JUnit in single-threaded execution and introduces negligible overhead (Section 4).

Comparisons The two most widely used unit testing frameworks for Java are JUnit and TestNG. While TestNG provides some features that JUnit does not offer, such as dependent and data-driven tests, neither of the two frameworks includes any additional support for addressing the problems posed by concurrency.

Recently, both JUnit and TestNG gained the ability to run multiple tests, or multiple instances of the same test, in parallel. The libraries jconch [4] and parallel-junit [5] add this feature to older versions of JUnit. Running tests in parallel can shorten the testing time on multi-core machines and in some cases reveal bugs that only occur during concurrent execution. These parallel extensions, however, still ignore the fundamental flaws of JUnit and TestNG in detecting errors in multi-threaded code, such as uncaught exceptions in spawned threads.

2. UNCAUGHT EXCEPTIONS
When a Java program throws an exception, the Java Virtual Machine (JVM) unwinds the stack of the thread in which the exception was thrown until a suitable catch block is found. If no such catch block exists and the stack unwinds completely, the thread is terminated. Unit testing frameworks for Java employ a `catch(Throwable t)` block to detect uncaught exceptions in the main test thread and report failure. Since test assertions in these frameworks are implemented using exceptions, our discussion of uncaught exceptions also covers failed assertions.

This catch block only applies to the test's main thread. Since Java threads by default do not have uncaught exception handlers installed, exceptions thrown in other threads are ignored. Listing 1 contains a JUnit 3.8.2 test case that demonstrates this.

Concurrency is ubiquitous in Java programs because multiple threads are required to support responsive user interfaces. Nearly all non-trivial applications with a GUI (graphical user interface) involve multi-threading. GUI frameworks like AWT/Swing and SWT rely on an event-handling thread to process all GUI input events and to access and update GUI components. The contracts for most AWT/Swing and SWT methods stipulate that the method must be executed in the event-handling thread. Hence, unit tests that manipulate GUI components (e.g., creating and modifying Swing documents) must run some code in the event-handling thread. In fact, essentially all non-trivial method calls on these objects must run in the event thread. If a call on a GUI component method in the event thread is erroneous, the method may throw an exception indicating an error, but JUnit completely ignores this exception and reports success as long as no exceptions are thrown in the main thread. Similarly, a JUnit test may attach a listener to a GUI component (e.g., add a `DocumentListener` to a `Document`) to perform tests whenever the listener is fired. Even when such a listener explicitly calls the `fail` method, JUnit will not report failure because the exception generated by the call is not thrown in the main thread. (The listener is executed as a postlude to calling a method in the corresponding GUI component, which must be done in the event thread.)

Our modified ConcJUnit framework creates a new thread group with an overridden `uncaughtException` method, and then creates a thread in this group. The test is executed in the new thread while the framework waits for it to finish. If an exception is thrown in the test's main thread or any of its child threads, the test group's `uncaughtException` method is invoked. It stores information about the uncaught exception and makes it available to ConcJUnit. When the test has ended, ConcJUnit retrieves the information from the thread group and declares the test a failure if any thread was terminated by an exception.

The use of a thread group is essential because a new child thread inherits its parent's thread group (unless a specific thread group is passed to the child's constructor); therefore, uncaught exceptions in child threads also invoke the overridden `uncaughtException` method, recording an error that will force the test to fail.

Java 5.0 introduced the `setDefaultUncaughtExceptionHandler` method, a mechanism for defining a default exception handler method that is associated with every thread. We decided to use thread groups nonetheless as they offer backward compatibility and robustness: Since thread groups in Java are hierarchical, we can still test programs that use thread groups themselves. On the other hand, there is only one default uncaught exception handler, and if a program removes ConcJUnit's handler, uncaught exceptions would no longer be processed correctly.

3. ENFORCED JOINS

3.1 Child Thread Outlives Test

Listing 1 exhibits another flaw, one that can also lead to a successful test, even though an uncaught exception is thrown: There is no guarantee that the child thread will reach the point of

```
import junit.framework.TestCase;

public class TestInOther extends TestCase {
  public void testException() {
    new Thread(new Runnable() {
      public void run() {
        // should cause failure but does not
        throw new RuntimeException();
      }
    }).start();
  }
}
```

Listing 1: Uncaught exception in other thread

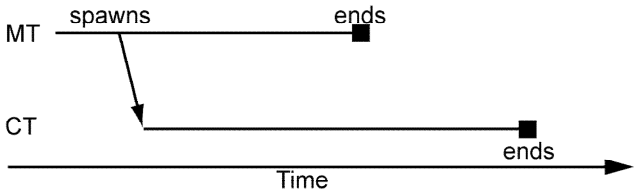

Figure 1: Child thread CT outlives test's main thread MT ("no join" warning)

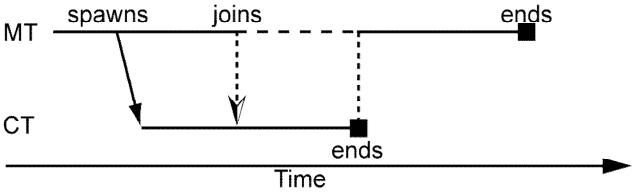

Figure 2: Main thread MT joins with child thread CT

```
import junit.framework.TestCase;

public class TestInOther extends TestCase {
  public void testException() {
    Thread child = new Thread(new Runnable() {
      public void run() {
        // exception detected with ConcJUnit
        throw new RuntimeException();
      }
    });
    child.start();
    while(child.isAlive) {
      try {
        child.join(); // wait until child done
      }
      catch(InterruptedException ie) {
        // interrupted while waiting
        // child may not be done yet
      }
    }
  }
}
```

Listing 2: Main thread waits for child thread to complete

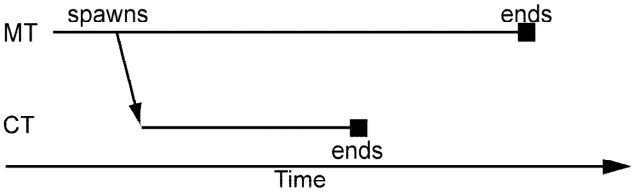

Figure 3: Child thread CT ends before main thread MT, but without join ("lucky" warning)

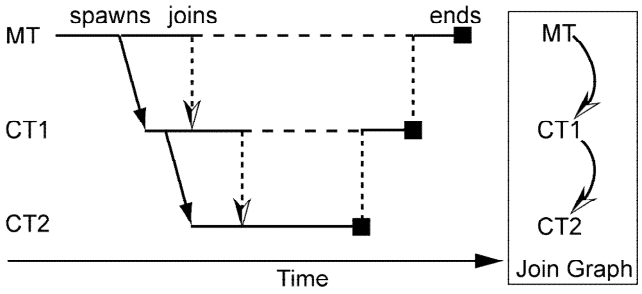

Figure 4: Each parent thread joins with its child thread (MT joins with CT1, CT1 with CT2)

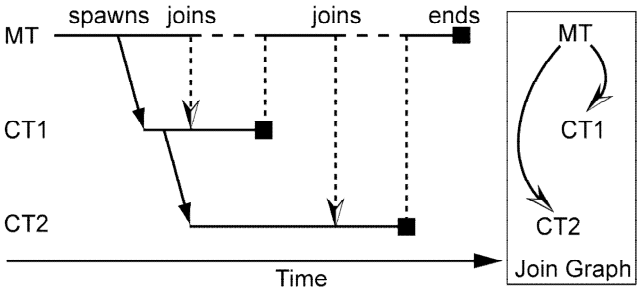

Figure 5: Main thread joins with both child threads (MT joins with CT1, MT with CT2)

failure before the main thread has finished and the test ends. This situation is depicted in Figure 1.

A correctly written test ensures that all child threads have terminated before the test ends, guaranteeing that the test is aware of any uncaught exceptions thrown in child threads before the test result is determined. Java's `Thread.join` method can be used to suspend the test's main thread until a spawned child thread has finished executing. Figure 2 displays the behavior of a correctly written test. The source code for such a test can be found in Listing 2.

To increase the framework's ability to detect badly written tests, ConcJUnit enumerates all living threads in the test's thread group when the test has ended. If threads are still found to be alive, a "no join" warning is emitted. Some system threads and daemon threads are excluded from this check, permitting them to outlive the test's main thread. These threads may have been created without the developer's knowledge and are terminated automatically at the end of the application's runtime. They only remain alive in a unit test because unit tests are executed in series within the same JVM.

3.2 Child Thread Terminates Without Join

The check for living threads described in the previous section only emits warnings for a faulty test whose main tread terminates before all child threads have finished. A more common problem is that a test may fortuitously succeed even though it did nothing to enforce that the main thread finishes last. A test exhibiting this behavior is depicted in Figure 3.

A fork/join design in which each parent thread has to join with all of its child threads solves this problem. Figure 4 demonstrates this scheme: The main threat MT spawns a child thread CT1; CT1 itself spawns another child thread CT2. CT1 cannot terminate before CT2 has terminated, and MT cannot end before CT1 has ended. Therefore, MT cannot end before all of its ancestor threads have finished executing.

This simple model is common in the parallel algorithms literature, but it may be too restrictive for general-purpose Java programs. For example, it should be permissible for the main thread to join directly with all of its ancestor threads, whether it started them itself or not. This is illustrated in Figure 5. Another scenario ensuring that all ancestor threads terminate before the main thread is to spawn a chain of helper threads, each guaranteed to outlive the previous thread, and force the main thread to join with the last helper thread. This situation is shown in Figure 6.

The concept of a chain can be generalized into a directed acyclic graph called "join graph", initially just consisting of a node for the main thread. Each time a new child thread is spawned, a new node is added to the graph, and every time a thread A joins with another thread B, an edge from A to B is added. Such an edge indicates that B terminated before A. Therefore, to ascertain that all child threads have terminated before a test's main thread ends, we only need to verify that all nodes are reachable from the main thread's node in the join graph. Note that in Figure 4, Figure 5 and Figure 6 all nodes of the join graphs are reachable from the main thread's node MT. In Figure 7, however, where no thread joins with CT2, the node for CT2 is not reachable from MT, indicating that a "lucky" warning should be issued, asserting that the proper termination order is not guaranteed.

While the improvements described in the previous two sections only require changes to the JUnit framework, detecting child threads that were not targets of a join operation requires modifying the Java Runtime Environment (JRE). The bytecode of the `java.lang.Thread` class must be augmented to perform the necessary bookkeeping at the end of the `Thread.start` and `Thread.join` methods.

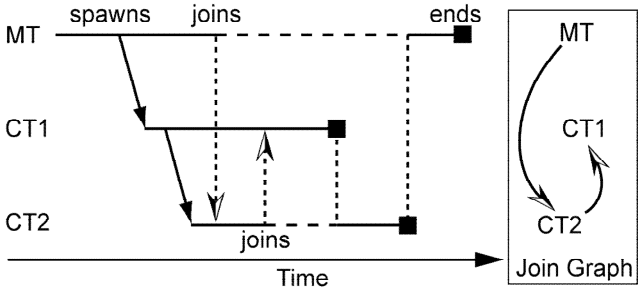

Figure 6: Main thread joins with last thread in chain (MT joins with CT2, CT2 with CT1)

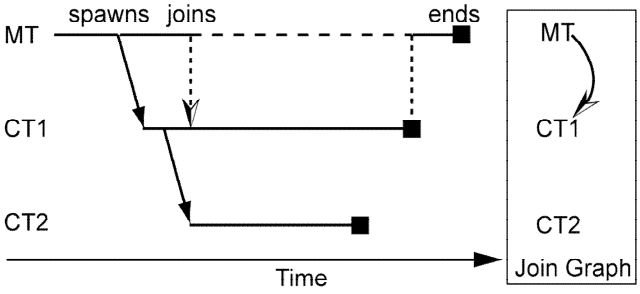

Figure 7: CT2 not reachable in join graph (MT joins with CT1, CT2 not joined by any thread)

ConcJUnit includes a tool that processes the rt.jar file (or classes.jar file on Mac OS X) of the JRE the user has installed, generating a replacement rt.jar file containing the modified `java.lang.Thread` class and its helpers. During testing, this replacement rt.jar is put on Java's boot classpath using the `-Xbootclasspath/p:rt.jar` command line option.

Similar to the way we construct the join graph, we also create a "start graph" that records the child threads spawned by each thread. The bookkeeping required to maintain the join and start graphs is performed using lock-free data structures to minimize the impact on the thread scheduling of the test.

At the end of a test, ConcJUnit attempts to retrieve the contents of these graphs using reflection. The library is not hard-linked against the modified `java.lang.Thread` class and therefore also works without it on the boot classpath; in that case, ConcJUnit just does not emit "lucky" warnings. Checking whether all child threads ended without being joined is efficiently implemented using set differences. ConcJUnit computes *S*, the set of threads reachable from MT in the start graph, and *J*, the set of threads reachable from MT in the join graph. If the difference $S - J$ is non-empty, then some child threads were not required to end before the main thread ended, and a "lucky" warning is generated.

4. ANALYSIS

To test the effectiveness and performance of ConcJUnit, we replaced JUnit with ConcJUnit and executed the unit test suites for DrJava [6] (revision 4918), an integrated development environment for Java, and JFreeChart [7] (1.0.13), an open-source library to display data visually. The extensive JFreeChart tests were not concurrent, but they all passed, demonstrating the compatibility of ConcJUnit with existing code.

Of the 900 unit tests contained in the DrJava test suite, 880 tests passed without any warnings. A single test emitted a "no join" warning, and 18 tests issued "lucky" warnings regarding their join behaviors. There were no tests that failed as a result of replacing the unit testing libraries, but one test timed out.

Upon examination of the source code, the 18 "lucky" warnings and the single "no join" warning all turned out to be legitimate. The "no join" warning was issued during a test that created a remote process which did not terminate during the test, causing the thread waiting for the termination to outlive the test.

The "lucky" warnings were emitted by tests that in fact did not join with all the child threads they had spawned. Instead, they used a wait-notify scheme to ensure that the child threads terminate before the tests end. In all of these cases, the developers had taken care that there were no more lengthy operations after the notifications, and that an uncaught exception after the call to notify was unlikely. This practically makes the wait-notify scheme equivalent to a join. However, if additional work were to be performed after the notification, and if one of the operations were to fail, such a test could be incorrectly declared a success.

We did not discover any tests that ignored uncaught exceptions or failed assertions in spawned threads, but for DrJava, a mature project built with test-driven methods, this was expected. Using ConcJUnit allowed us to replace the handler for uncaught exceptions that was custom-built for DrJava with the general one found in ConcJUnit. Doing this also made a test of the exception handler redundant, eliminating one of the "lucky" warnings.

The overhead, introduced by ConcJUnit to handle uncaught exceptions in all threads and to detect the "no join" and "lucky" conditions, was negligible. The total slowdown for ten runs of the entire test suite was 55.2 seconds, or 1.1 percent of the 5252.4 seconds it took to run the entire suite ten times using JUnit.

5. FUTURE WORK

Our improvements to JUnit will not detect all uncaught exceptions that *could* occur; only the uncaught exceptions thrown in the chosen thread schedule are found. Similarly, ConcJUnit will not report all threads that *could* end without being joined by the main thread; it will only emit warnings for those threads that actually ended in such a way. Even if the test suite passes ConcJUnit without failures or warnings, it is still possible that the unit tests fails during the next run.

Unit testing requires that known input generates known output, and that the entire computation is deterministic, and ConcJUnit does not address the important issue in potential non-determinism in multi-threaded unit tests. A truly comprehensive unit testing framework should provide practical tools for assuring that concurrent unit tests are deterministic. One possible approach is to provide tools for generating and replaying representative schedules for each test method in a test class while running a data race detector such as Eraser [8] in parallel.

6. CONCLUSION

Unit testing concurrent programs is difficult for two reasons: inadequate existing frameworks and non-deterministic thread scheduling. ConcJUnit, the extension of JUnit introduced in this paper, improves JUnit by detecting uncaught exceptions and failed assertions in all threads and by warning the developer if a child thread could outlive the test's main thread because no join operation was performed. While this does not lead to deterministic unit test results for concurrent programs, ConcJUnit remedies two of the most serious problems in existing frameworks. Designed as drop-in replacement for JUnit, our framework is immediately applicable for Java programs and provides important foundations for future extensions.

ConcJUnit is compatible with all three major platforms: Windows, Linux, and Mac OS X. The project is open source and available at http://www.concutest.org/

7. REFERENCES

[1] Jefferies, Ron. http://www.xprogramming.com/

[2] JUnit Project. *JUnit*, http://junit.org/

[3] TestNG Project. *TestNG*, http://testng.org/

[4] Fischer, Robert. *jconch*, http://code.google.com/p/jconch/

[5] Kawaguchi, Kohsuke. *Parallel-JUnit*, https://parallel-junit.dev.java.net/

[6] Rice JavaPLT. *DrJava*, http://drjava.org/

[7] JFree.org Project. *JFreeChart*, http://www.jfree.org/jfreechart/

[8] Savage, Stevan, et al. *Eraser: A Dynamic Data Race Detector for Multithreaded Programs*, ACM Trans. Comput. Syst., 15, ACM Press, New York, 1997. No 4, p. 391-41