-- Geoff Groos, Vivekumar Patel, Rafael Bradley

This document is available online, with embedded links, here, or at http://bit.ly/introopder

Abstract

Writing concurrency tests is enormously difficult: aside from the lack of expressiveness around parallelism and concurrency in popular testing frameworks, finding and laying-out the setup required to push a program into a state where it may violate atomicity rules is difficult. This makes a completely automated detection and test-generation scheme for such a problem very attractive. We explore combining two existing frameworks, one test-generation software, Randoop, and another atomicity exploration software, Intruder, in an attempt to create a single automatic concurrency-testing method, requiring only the binaries of the component under test. We show that with this approach one can easily discover some atomicity problems.

Introduction

Writing tests that assert on the correct use of locks, monitors, immutable objects and thread-confinement is very difficult.

Below are two tests against the same component:

- The first is a test that attempts to assert that the component under test adequately implements a particular piece of functionality. Notice the brevity.
- The second is a test that attempts to assert that that same code works given a specific --previously problematic-- interleaving of two methods. Notice the length and increased complexity of the code.

```
  //////////////////////////////////////////////////
  // Test 1

  @Test
  public void
when_marshaller_is_in_idle_state_followed_by_running_state_determineIfOptimizationC
ontinues_should_recurse_once_and_return_true(){
     //setup
     OptimizerSynchronizationMarshaller marshaller = makeMarshaller();
     when(stateMachine.getState()).thenReturn(State.Pending, State.Running);

     //act
     boolean shouldContinue = marshaller.determineIfOptimizationContinues();

     //assert
     assertThat(shouldContinue).isTrue();
     assertThat(marshaller.acknowledgements).containsExactly(pendingState);
     verify(stateMachine, twice()).getState();
  }

  /////////////////////////////////////////////////
  // Test 2
  /**
   * This is a test to assert that the scheme handles contention between the
optimizer and the HMI thread
   * in the Start-Order-Queue properly. This test attempts to assert that there is
no race.
   * (first order to be recieved -- more specifically the first one to lock the
state machine)
   * between these two threads. This test asserts that, in one scenario, the locks
work out correctly.
   */
  @Test(timeout = 1000)
  public void
when_in_state_StartPending_and_a_StopOrder_happens_immediately_before_the_optimizer
_tries_to_ack_should_become_idle() throws InterruptedException {
      //setup
      marshaller = makeMarshaller();
      marshaller.getStateMachine().currentState = StartPending;

      CountDownLatch stopOrderShouldContinueSignal = new CountDownLatch(1);
      CountDownLatch stopOrderIsReadyToContinueSignal = new CountDownLatch(1);
      CountDownLatch optimizerIsChecking = new CountDownLatch(1);

      LinqingList<Transfer> reportedTransfers = new LinqingList<>();

      marshaller.getStateMachine().addTransferListener((oldState, xfer, newState)
-> {

          reportedTransfers.add(xfer);

          if(xfer == Transfer.StopOrder){
              //blocks us while we're in the middle of transferring, a vary
precarious spot!
              stopOrderIsReadyToContinueSignal.countDown();

ExceptionUtilities.failOnException(stopOrderShouldContinueSignal::await);
          }
      });

      Runnable hmiWorkload = () -> {
          marshaller.issueOrderAndAwaitAcknowledgement(Transfer.StopOrder);
      };

      Runnable optimizerWorkload = () -> {
          optimizerIsChecking.countDown();
          boolean shouldContinue = marshaller.determineIfOptimizationContinues();
          assertThat(shouldContinue).isFalse();
      };

      //act I : get HMI thread ordering change-back to stop
      Thread fauxHMIThread = syncingUtilities.asynchronously("fauxHMIThread",
ThreadTag.HMI, hmiWorkload).get();
      stopOrderIsReadyToContinueSignal.await();

      //act II : get optimizer asking if it should continue while HMI thread is
transferring states
      Thread fauxOptimizer = syncingUtilities.asynchronously("fauxOptimizer",
ThreadTag.Optimizer, optimizerWorkload).get();
```

```
      optimizerIsChecking.await();
      syncingUtilities.sleepUnlessInterruptedFor(CheckDelay); //let optimizer
*attempt* to continue
      Thread.State optimizerStateAfterChecking = fauxOptimizer.getState();

      //act III : let the HMI thread finish its transfer
      stopOrderShouldContinueSignal.countDown();
      fauxHMIThread.join();
      fauxOptimizer.join();

      //assert
      assertThat(marshaller.getState()).isEqualTo(Idle);
      assertThat(reportedTransfers).containsExactly(StopOrder);
      assertThat(optimizerStateAfterChecking).isEqualTo(Thread.State.BLOCKED);
      eventBus.shouldNotHaveBeenAskedToPost(OptimizationRunGroupHaltedEvent.class);
//remember this test is pending -> stopped
  }
}
```

Such tests require a significant amount of time to write and consist of more complex constructs than even the code they're trying to test. Such difficulty and complexity has led to a number of researchers investigating the possibility of leveraging dynamic code analysis to programmatically find the errors the above test is seeking, given a minimal driver.

Enter Intruder, a tool that attempts to augment a traditional functional testing suite with atomicity violation detection and a system to generate tests that push a particular class into a state wherein it is likely to encounter a failure pertaining to the atomicity of its data. This tool dynamically analyses an *existing* set of functional tests, inspecting those tests access patterns in an attempt to find atomicity violations. Such a tool enables substantially better testing of the concurrency properties of a given class, but can require an extensive set of existing functional tests to be effective. Further, functional tests have their own set of biases and may not accurately reflect the actual use of the class in production, meaning using existing test suites may not enable Intruder to reveal all concurrency problems in a class.

Thus it would be very desirable to use a tool like Randoop to generate a set of tests for use with Intruder. Using Randoop in this role allows a programmer to very quickly generate a suite of tests that covers many (or all) of the public methods on a class and *likely* emulate the kinds of uses the class under test is likely to see in production. These uses will reproduce access patterns for Intruder to analyse.

Our goal is to use Randoop as a front end functional-test-generation scheme for Intruder, so that: - the tool can be run without any existing tests, given only the class or jar files of the component under test and - the tool avoids the bias of existing functional tests, testing *possible* access patterns rather than only those thought of by the author of the functional tests.

Background

In the discussion for Intruder, the authors mentioned future work in using test generation schemes to drive Intruder.

> Obviously, the quality of the multi-threaded tests is dependent on the input sequential seed testsuite. For example, if the code pertaining to an atomicity violation is not covered by the sequential test, our approach will be unable to synthesize a multithreaded test. Apart from developing a manual sequential seed testsuite as described above, we can also generate these testsuites using automatic test generators [including randoop].

We decided to explore the use of Randoop with Intruder. Unfortunately we found that the way they use Randoop was both not documented and not obvious; at best its inclusion in their project was as a build-time switch, something that we don't have access to since we can only work with their included binaries.

Given this, we wanted to see what would be required to create an end-to-end atomicity detection and test-generation scheme, given only the binaries for

Randoop

Randoop was conceived by Carlos Pacheco and Michael D. Ernst in 2005, and has since been adopted by several people at Microsoft and developed into a mature and well-used tool. Its ability to very quickly generate a large set of reasonably high quality tests makes it attractive for those looking to write tests against an extensive API or object. Randoop uses an iterative process to build a sequence of method calls on an objects public interface, not unlike those the object is likely to see in production. Then, it iterates on the results and tries to re-use them in order to invoke more complex sequences that uses different sets of method calls. This approach has led developers to reveal unknown issues even in extensively used libraries such as Oracle JDK.

We use Randoop because it quickly generates a reasonably comprehensive set of tests (regression tests in this case) which can be used to drive Intruder and, therefore, find concurrency issues with little to no human intervention. Because the strategy of intruder is to build sequences of public method calls, it is likely to expose access patterns used by users of the code under test.

Intruder

The access patterns of a particular object are critical to the functionality of Intruder; Intruder is a tool that analyses existing test suites to discover violations of presumed atomic properties of objects. These atomic properties largely include the update of particular fields. An obvious example of an atomicity violation would be an unguarded use of a unary integer-increment:

```
class POJO{

  int x;

  public void incrementX(){ x++; }
}
```

In such a case it is possible that two calls to `incrementX()` on different threads clobber each other, resulting in each thread reading the same value of `x`, and writing back that value of `x` incremented once, meaning that despite two different calls to `incrementX()` returning, `x`'s value only goes up by one, not two as would be expected. Modifying `incrementX()` to acquire a monitor-lock with synchronization, by replacing the signature with `public synchronized void incrementX()` would suffice in this example because the method does not take any arguments and does not read from any external sources. However, if the amount `x` was incremented was dependent on an argument or a constructor-supplied field, this level of locking would likely be insufficient, as it would not prevent another thread from modifying the dependant. These kinds of atomicity violations are the ones Intruder targets.

Intruder attempts to discover possible problematic methods and method-arguments by instrumenting code loaded by a supplied test suite (--which contains the code under test). This instrumentation carefully tracks the sequence of field accesses and mutations, and what locks were acquired during any one action.

Because Intruder's model does not include timings or tracking of concurrency primitives (such as threads and executors), a large class of problems are not directly detected by it, such as race conditions and deadlock. Even for problems directly relating to field access and mutation, because of limitations relating to tracking of particular instances and the reliance on a driver to properly induce the problematic states, Intruder is by no means a complete system for discovering problems. However, for the large class of problems around staleness of data, assuming a test suite can adequately represent the majority of access patterns to their tested objects, Intruder is effective at finding atomicity violations.

Intruder generates a suite of multithreaded tests to prove the presence of atomicity violations with a tool called CTrigger. The generated multithreaded tests are not traditional in a red-bar-green-bar sense, as they must themselves be analysed with a tool to reveal defects, thus the usefulness of the tests generated by Intruder is limited. However, by building CTrigger into the Intruder runtime, setting the flag `CTRIGGER` to `ON` will cause Intruder to generate a text-based summary of the atomicity violations it detects, at the expense of significantly more time to run.,

Implementation

Our initial strategy was to simply execute Randoop on the libraries from Intruder's Table 5 (below), and pipe the results from Randoop into Intruder

Intruder Table 5: Benchmark Information

Benchmark	Version	Class
Colt	1.2	`DynamicBin1D`
OpenJDK	1.7	`StringBuffer`
OpenJDK	1.7	`Vector`

Benchmark	Version	Class
Carbonado	1.2.3	`SkipCursor`
Cometd	2.7.0	`TimesyncClientExtension`
eXo	3.8.2	`ApplicationStatistic`
eXo	3.8.2	`PortalStatistic`
Batik	1.7	`CompositeGraphicsNode`
OpenNLP	1.5.3	`PerformanceMonitor`

This can be implemented with the following pseudo-code

```
for testFramework in $intruder_table5
  randoopTests = randoop --input $testFramework
  adaptedTests = adapter --input $randoopTests
  compiledRandoopTests = javac $adaptedTests
  intruderReport = intruder --input $compiledRandoopTests
  echo $intruderReport >> summary.txt
```

To accomplish this, we would need to: - create a script to run the above code - write an adapter, which we named porter, to convert the output from Randoop to an input - configured `javac` to handle the various dependencies at the various stages of compilation - find matching atomicity violations detected by Randoop and the existing functional test scheme, map violations to defects.

A complete implementation of this script can be found in runner.sh

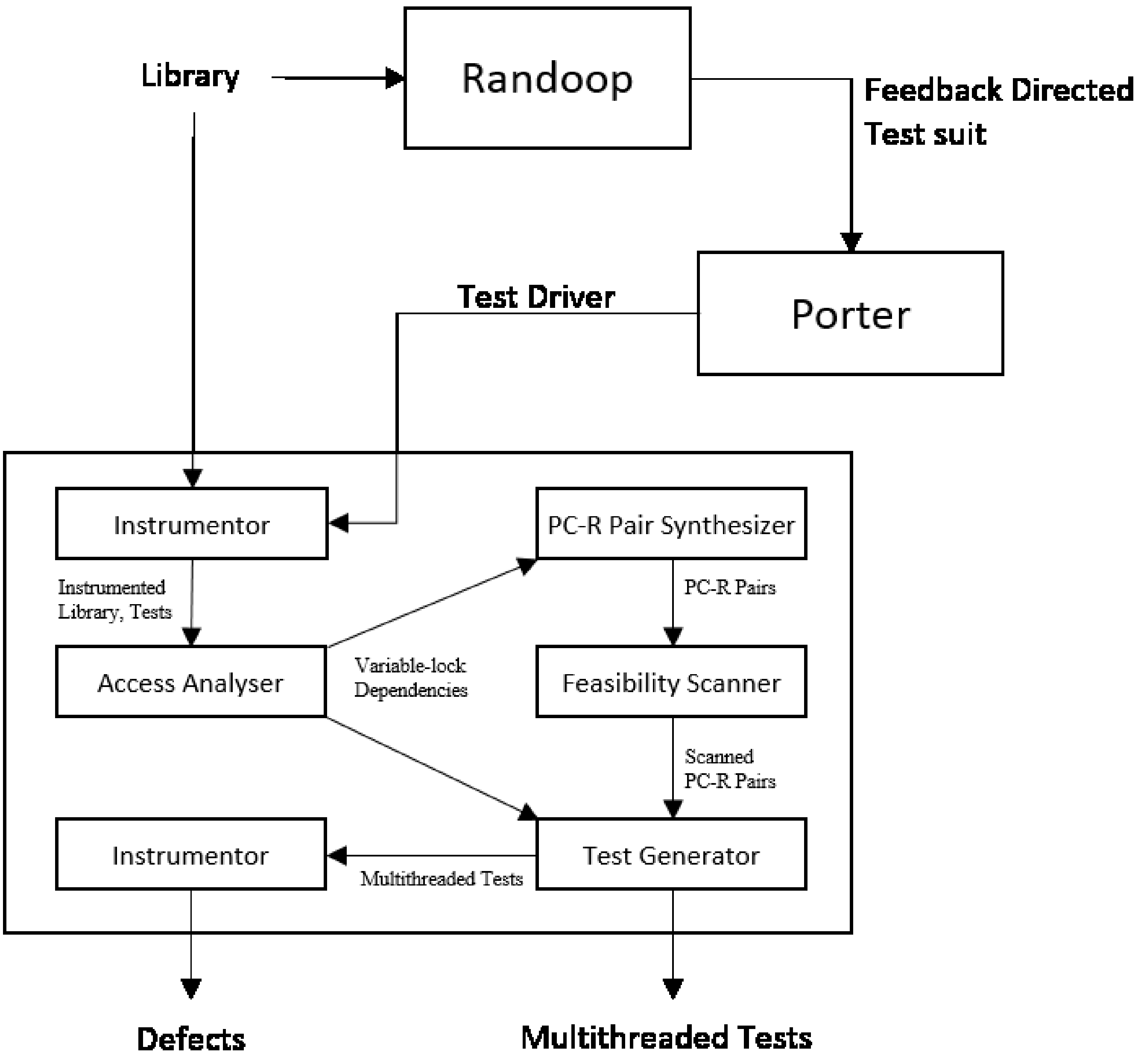
The single biggest technical obstacle to driving Intruder with Randoop generated tests is that Randoop expects the results to be driven by JUnit and Intruder expects to have a single static `main` entry point. Unfortunately because of the analysis strategy of intruder we were afraid that we might see artifacts if we simply wrote a `main` method wrapping the JUnit API to drive the tests created by Randoop. To overcome this we wrote a small Java utility, `porter`, that generates a `main` method wrapping target JUnit test methods created by Randoop.

The next problem would be to ensure that the targeted component for test and its dependencies were available as necessary. This proved to be consistently difficult, as we couldn't find a single solution to manage dependencies for all of the target test suites.

Finally we needed to catalogue the discovered violations, and map them to defects. This also proved very difficult as we quickly realized you had to have more knowledge about the tested frameworks than any of us had to come to any educated conclusions about the results from intruder. Thus we ignored this step for the bulk of our testing instead simply focusing on the count of defects discovered in the various testing schemes.

The resulting flow is described in figure 1.

Figure 1: augmented architecture of Intruder with Randoop and Porter



Experiments

Our experiments were performed on an Kubuntu 15.10 dual-core 2Ghz virtual machine with 4GB RAM.

Using our script in conjunction with manually downloading and configuring dependencies, we were able to execute Randoop and Intruder on 2 of the 9 records in Intruder Table 5.

Despite a number of attempts we were unable to download find, configure, and build all of tests specified in Intruder Table 5. Finding the correct version of a library, the correct version of the JVM to go with that library, all of each libraries dependencies, the build systems required by each library, and so on overwhelmed us. With more time and deeper integration with a dependency management tool, we likely would be able to find more success.

As an alternative, and to establish a baseline, Intruder includes a number of simple but illustrative functional tests. As a proof-of-concept for our system we re-tooled to emphasize running these tests. Our method was consistent with Intruder's original results where no atomicity violations were found in the baseline tests: Intruder-functional-test-1, 3, 5, 9, 10, 11, 12 and 13. However, a higher execution time was observed due to the increased complexity of the test cases generated by Randoop.

Table 1 shows a comparison between the results obtained using hand-crafted test cases and the results obtained using our automated framework to obtain the test cases. All tests were run with CTRIGGER on.

Table 1: A comparison of Randoop-driven Intruder and manually-written-test-driven Intruder

> **Legend** - a: the intruder execution time in milliseconds. - b: the total number of multithreaded tests generated by Intruder - c: the amount of atomicity violations found

Test	Manually-Written (a, b, c)	Randoop (a, b, c)
intruder-funcitonal-test-1	0	0
intruder-funcitonal-test-2	1949, 1, 1	*error while running intruder*
intruder-funcitonal-test-3	0,	0
intruder-funcitonal-test-4	1838, 1, 3	2321, 2, 5
intruder-funcitonal-test-5	0	0
intruder-funcitonal-test-6	2216, 1, 1	2791, 1, 1
intruder-funcitonal-test-7	2504, 1, 1	*Randoop Unable to Generate Test*
intruder-funcitonal-test-8	2633, 1, 1	3242, 1, 0
intruder-funcitonal-test-9	0	0
intruder-funcitonal-test-10	0	0
intruder-funcitonal-test-11	0	0
intruder-funcitonal-test-12	0	0
intruder-funcitonal-test-13	0	0
Colt, `DynamicBin1D`	159744, 9, 27	97559, 8, 19
Batik, `CompositeGraphicsNode`	35959, 7, 47	21217, 2, 2
Batik, `CompositeGraphicsNode`	35959, 7, 47	21441, 10, 7

Due to the nature of intruder-functional-test-2, which had only one zero-argument method that did not call any other functions implemented, Randoop was not able to generate any test cases.

We also compared the functionality on two of the benchmark provided by Intruder. In both cases, after running intruder using the test driver generated by porter, we found a smaller subset of atomicity violations on the tested libraries, but also a shorter execution time.

Conclusion

From our work we found the need for automatically generated test cases, especially feedback-directed random test generation in this experiment, to detect atomicity violations using

Intruder in thread-safe libraries and components. We designed an approach that is less time consuming than the actual case in which the sequential test cases are generated manually. We synthesized a tool call `porter` and tested several small programs with and without bugs and two real world libraries using it. For all but one of those cases, our approach finds at least one of the underlying concurrency bugs. However, our approach generated more duplicates than the one described in Intruder because of several calls to same function.

To conclude we can say that the approach described in the current paper will be more useful in case when the tester doesn't know the underlying implementation and details about the library to be tested, but will also require substantially more development time. We hope that this system could represent a useful step in automation of concurrency bug detection.

Note about detected bugs

The bugs intruder detected related to failure in the scope of the locks and monitors acquired by components under test. In the Motivation section for Intruder (Intruder Figure 1) requires the use of the mutable collection variables that are not sufficiently synchronized. This can be solved with a more elaborate use of locks or use of immutable data structures.

Intruder Figure 1: Motivating example

```
DynamicBin1D.java:
------------------
```

```
synchronized DynamicBin1D sampleBootstrap(DynamicBin1D other, ..., BinBinFunction1D
function) {
// since "resamples" can be quite large, we care about performance and memory
  int maxCapacity = 1000;
  int s1 = size();
  int s2 = other.size();
  DynamicBin1D sample2 = new DynamicBin1D();
  cern.colt.buffer.DoubleBuffer buffer2 =
sample2.buffered(Math.min(maxCapacity,s2));
  // resampling steps
  for (int i=resamples; --i >= 0; ) {
    sample1.clear();
    sample2.clear();
    this.sample(s1,true,randomGenerator,buffer1);
    other.sample(s2,true,randomGenerator,buffer2);
    bootBuffer.add(function.apply(sample1,sample2));
  }
}
synchronized void sample(int n, boolean withReplacement, ...) {
  if (!withReplacement) { // without

  }
  else { // with
    Uniform uniform = new Uniform(randomGenerator);
    int s = size();
    for (int i=n; --i >= 0;) {
        buffer.add(this.elements.getQuick(uniform.nextIntFromTo(0,s-1)));
      }
    }
  }
synchronized void clear() {
  if (this.elements != null) this.elements.clear();
}
```

```
DoubleArrayList.java:
---------------------
```

```
/* You should only use this method when you are absolutely sure that the index is
without bounds.*/
public double getQuick(int index){
  return elements[index];
}
```

Intruder's discovered atomicity violation relates to the interleaving

Thread	calledCode
A	`sampleBootstrap(x)`
A	`x.size()`
B	`x.clear()`
A	`x.sample()` --error, x is now empty

This can be solved by some effort on behalf of the programmer by acquiring a more expansive lock, possibly including a mutex on the collection `x` (the first argument to the `sampleBootstrap` method). Without great care, the resulting locking scheme is likely to provide grounds for deadlock.

This problem can also be trivially solved by use of an immutable and persistent collection, such as those available in the standard libraries of Clojure and Scala. In general, referentially transparent (aka 'pure') data structures are immune to the kinds problems that can be discovered by intruder, as without any mutations there is simply no possibility of a write to provide for stale data.

References

Project Proposal

Intruder Samak, M., & Ramanathan, M. K. Synthesizing Tests for Detecting Atomicity Violations. Bergamo, Italy: ESEC/FSE 2015.

Randoop Pacheco, C et al. Feedback-directed Random Test Generation. Minneapolis, MN, USA: ICSE 2007.

CTrigger Park et al. CTrigger: Exposing Atomicity Violation Bugs from Their Hiding Places. New York, NY, USA: ASPLOS 2009