Unit Testing Concurrent Java Programs

Suma Shivaprasad
Staff Engineer,
Inmobi India, Bangalore

Nanditha Prasad
Assistant Professor, Department of Computer Science
Government Science College, Bangalore

ABSTRACT

Conventional unit-testing practices focus on testing small units of programs sequentially and are very likely to miss concurrent bugs such as race conditions, deadlocks and memory inconsistencies even when done extensively. They are not suitable for unit testing multithreaded programs.

This paper outlines guidelines for writing effective unit tests for concurrent Java programs. It also explores and compares the frameworks available currently for writing such tests. The most widely used unit testing frameworks for Java - TestNG and JUnit - do not provide good support for testing concurrent issues. Other frameworks such as MultiThreadedTC and Concurrency Analyzer allow the coordination of unit test's threads to produce a specific scheduling. However, it is the responsibility of the developer to test for all possible interleavings and scheduling of threads to unearth existing bugs and hence they are a deterministic way of approaching the problem.

This paper presents an alternate approach to the problem by integrating TestNG with the Java Path Finder(JPF) software model checker. JPF can be used to identify all possible interleavings of threads across execution paths to non deterministically detect concurrent bugs. In addition to this, it detects deadlocks by checking if all the threads have reached a blocked state at any point of execution. Adoption of such practices have helped in reducing concurrency related issues in our platforms to a great extent. It has helped in identifying issues early on in the development cycle and better reliability. Many open source platforms such as Ehcache run concurrent unit tests as part of their development process to maintain code quality. This, augmented with stress testing concurrency tools greatly help in improving the quality of code.

GENERAL TERMS

Software Engineering, Unit Testing, Concurrent Unit Testing

KEYWORDS

Concurrent Programming, TestNG, Java Path Finder, JUnit, Model checking, Thread scheduling, Deadlocks, Concurrent Unit Testing, ConAN, MultithreadedTC, ConTest.

1. INTRODUCTION

With usage of multi-core processors, concurrent programming is getting prominence. Writing good concurrent unit tests is as hard as writing good concurrent programs. Multithreaded programs are prone to concurrency bugs that depend on timing and scheduling of the program's threads. Common concurrency issues encountered such as deadlocks, livelocks, memory inconsistencies, race conditions are found only in system tests, functional tests, or by the user. Conventional unit testing is unsuitable for detecting concurrency bugs in multi-threaded programs due to their non-deterministic nature. They run sequentially with only one thread executing the code. The probability of uncovering a concurrency bug by running the test once is low. Other approach is to run these test cases many times.

However running a unit test repeatedly without enforcing different scheduling is not efficient enough to reveal concurrency bugs. Unit tests are also free from I/O causing the scheduler to produce the same scheduling during repeated test runs.

不加控制重复运行测试揭示故障的效率很低。

For multithreaded programs, test inputs need to be varied and various code execution paths/state space to be explored considering temporal ordering of events. Tailoring thread count so that the number of runnable threads at any time is a small multiple of the processor count will often result in a more interesting variety of interleavings/schedulings. A concurrent unit test succeeds if all possible interleavings have been examined and all of them have produced the expected results. Otherwise, the test fails, either because of an "ordinary" bug or a "concurrency" bug. Although the number of possible schedulings of a program grows exponentially with the program's length, unit tests in general are very short. Therefore, the number of possible schedulings of a concurrent unit test is small enough to explore all of them.

对于多线程程序，需要改变测试输入，并考虑事件的时间顺序探索各种代码执行路径/状态空间

执行交错数量随程序长度指数增长

Compared to concurrency bug detection tools, concurrent unit testing offers some advantages [1, 2]. Most of these tools define certain correctness criteria about the program's synchronization. Eg., A potential data-race if not all accesses to a shared variable are protected by a common lock. These correctness criteria are often limited to specific kinds of concurrency bugs. Based on the run-time observations or the source code analysis, the tools Eg., FindBugs decide whether these correctness criteria are violated or not. Depending on the correctness criteria, concurrency bug detection tools may report false positives and leave concurrency bugs undetected. As opposed to that, concurrent unit testing does not use any pre-defined correctness criteria. Correctness criteria are specified by the developer of concurrent unit tests explicitly as assertions. Concurrent unit tests must fulfill these assertions in all possible schedulings. If a scheduling is found in which the developer's assertions are violated, and the result depends on the actual scheduling, then the test found a concurrency bug [3, 4].

This approach is more general, as it is not limited to a certain kinds of concurrency bugs, but can detect data races, atomicity violations, and order as well. Furthermore, concurrency bug detection tools usually require a running program for their run-time analysis, while concurrent unit testing requires only working units. Therefore, concurrent unit testing can be used earlier in the development process and can be integrated with Continuous Integration during commit builds.

2. GUIDELINES FOR DESIGNING CONCURRENT UNIT TESTS

Since testing concurrent code is difficult, designers are expected to spend more time designing and executing concurrent tests than spent for sequential ones. The guidelines to be considered when designing and running tests for concurrent programs are:

- **Tests are Probabilistic**: Finding concurrent bugs is probablistic given that the test cases are probabilistic as well. Run Tests for longer duration to increase the probability of finding concurrent bugs.

测试是概率性的

- **Explore more of the State Space**: Explore all the code paths with temporal considerations which includes relative orderings of events. For example, in a Bounded Queue test, explore all the relative timings of insertion and removal.

探索更多的状态空间

- **Explore more Interleavings:** Run multiple threads with preemptions introduced in synchronized blocks to increase the likelihood of finding race conditions and deadlocks.

探索更多的交错

- **Match Thread Count to the Platform**: Tailoring thread count so that the number of runnable threads at any time is a small multiple of the processor count will often result in a more interesting variety of interleavings.

线程数是处理器数量的一小倍

- **Avoid introducing Timing or Synchronization artifacts**: Concurrent data structures Eg. A shared queue requires synchronization when shared across threads. If unit test framework introduces its own synchronization, it might disturb the timing and scheduling of the tested component.

避免引入定时或同步构件

3. EXISTING CONCURRENT TESTING FRAMEWORKS

The most widely used unit testing frameworks for Java in companies worldwide are TestNG and JUnit. Although TestNG provides some features that JUnit doesn't, such as dependent and data driven tests, neither of the frameworks includes adequate support for addressing the problems posed by concurrency. They only facilitate parallel execution of the tests which does not add any benefit; they do not actively attempt to vary or influence the scheduling of threads. Hence, none of these tools are reliable enough to detect concurrency bugs, and cannot show that a concurrent test produces the expected results independently from the scheduling.

3.1 MultithreadedTC

MultithreadedTC is an opensource framework that allows a test designer to exercise a specific interleaving of threads framework in an application. It features a clock that allows test designers to coordinate threads even in the presence of blocking and timing issues. It can also detect deadlocks and livelocks. The main drawback is that the developer has to write a specific sequence of interleaving threads to test the system for concurrency conditions. The responsibility of determining the scheduling and interleaving of the threads totally rests on the developer. So, some of these combinations might get missed out [5].

3.2 ConAn

Concurrency Analyzer is a script-based test framework that, like MultithreadedTC, uses a clock to synchronize the actions in multiple threads. Again, the responsibility rests with the developer to determine the scheduling and interleaving of the threads while writing test cases and there are chances of them getting missed out [6].

3.3 ConTest

ConTest is an internal IBM Framework that works with existing tests and no code change is required. Basically it records and replays thread interleavings that lead to faults by manipulating bytecode. It uses sleep() and yield() to test different interleavings each time a test is run. It also detects deadlocks and provides synchronization coverage. Synchronization coverages measures how much contention exists among synchronized blocks and allows the developers to visualize whether they have covered interesting interleavings. Main disadvantages are that its algorithm also modifies the original program to add synchronization and it is proprietary [7, 8].

4. PROPOSED APPROACH

4.1 JPF and TestNG Integration

This approach proposes a concurrent unit testing framework which combines unit testing of TestNG and model checking of a framework called Java Path Finder(JPF) to detect concurrency bugs by exploring reachable code state space including all thread interleavings. 通过探索可达的代码空间包括所有的线程交错来检测故障

TestNG was chosen since it is recommended and used by the various companies throughout the world and provides better features than JUnit. This framework is an extension to TestNG and supports all existing functionality of TestNG [9, 10].

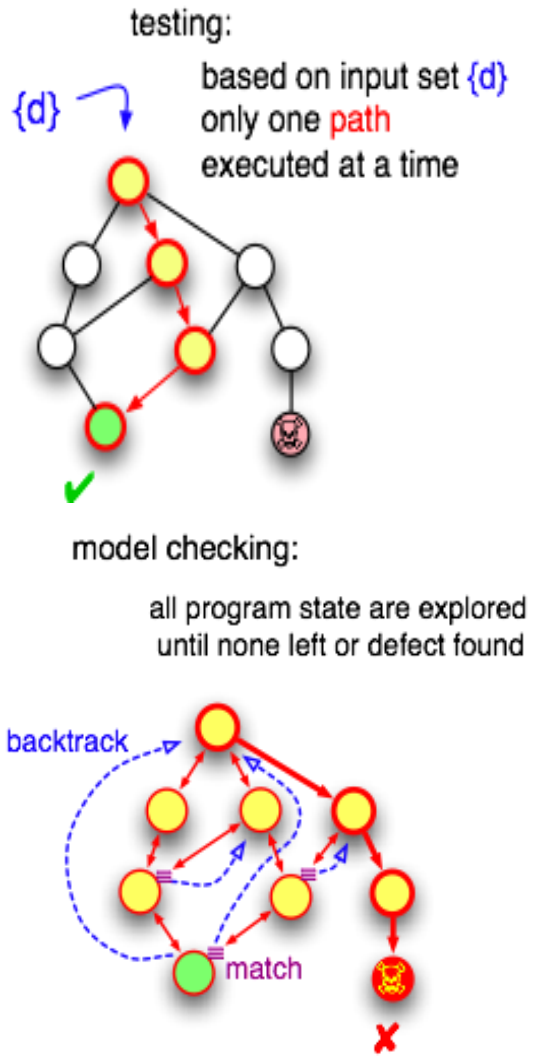

Figure 1 : Testing Vs Model Checking

Model checking is a formal method that exhaustively explores all possible system under test behaviours. For example, if we have a program that uses a sequence of random values: testing always processes just one set of values at a time, and we have little control over which ones. But Model checking does not stop until

it has checked all data combinations or has found an error as illustrated in Figure 1.

Mapping it to a concurrent programming example, all we know is that different scheduling sequences can lead to different program behavior (e.g. if there are data races), but there is little that can be done in conventional tests to force scheduling variation. There are program/test spec combinations which are "untestable".

传统测试中，强制线程调度几乎无能为力

JPF is an explicit state software model checker for Java bytecode which implements a backtrackable state tracking JVM and runs on top of a host JVM. Being a virtual machine, JPF model checker has complete control over all threads of our program, and can execute all scheduling combinations. It explores all possible execution paths of a java program without recompiling [11].

实现了跟踪JVM的执行状态

作为一个虚拟机，JPF模型检查器可以完全控制程序的所有线程，并可以执行所有的调度组合

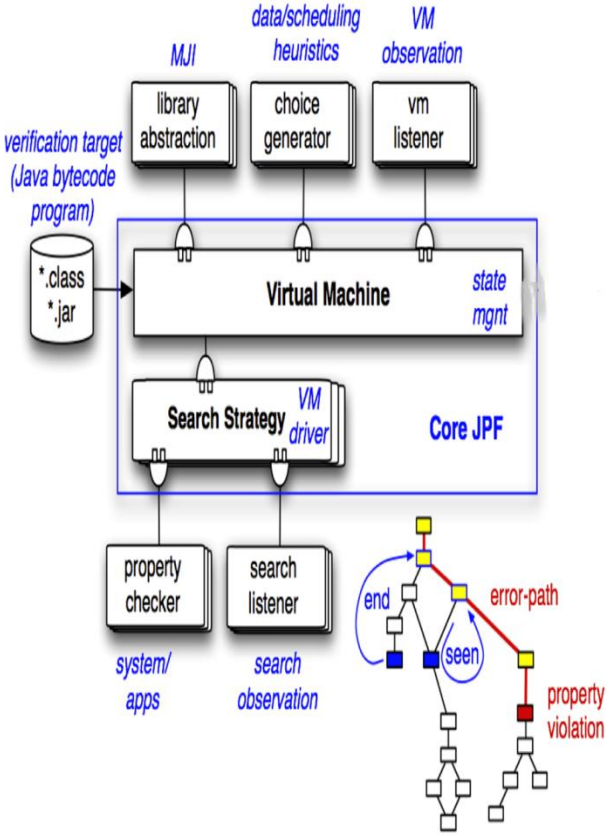

Figure 2:

JPF works only for < 10 KLOC programs. The formula below in Fig 2 illustrates the possible number of states given the number of threads(P1, P2, …Pn) , each thread having n_i atomic instruction sequences. For 2 threads with 2 atomic sections each this gives us 6 different scheduling combinations. For 8 sections the result is 12870, 16 sections yield 601080390 interleavings.

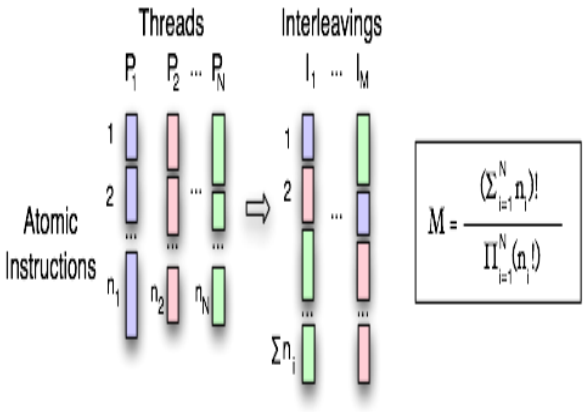

Figure 3 : Thread Interleavings

状态爆炸

Since unit tests are small units of code, state explosion is not a concern. Moreover, JPF uses partial order reduction to limit state space explosion which helps in reducing the execution time for unit tests. JPF also detects deadlocks by checking the state of all threads. If all the threads reach a blocked state, then they are presumed to be deadlocked.

Since JPF itself acts as a special JVM, it can only operate with Java applications i.e it requires a main() method. Unit tests generally do not have this and hence to bring JPF and unit tests together, a small wrapper program was developed which wraps the unit tests with a main() method. For each concurrent unit test, the framework invokes JPF to run the wrapper program with configurable options. The wrapper instantiates the test case class and invokes the test methods. The wrapper program is also responsible for starting up the threads and initializing the barrier with the given number of threads. JPF Output can be analyzed through its listener interfaces. Using this interface, the framework observes execution of concurrent unit tests and their failures. If an exception is thrown by the test, the wrapper checks if it is an unexpected exception or an appropriate one.

4.2 Architecture

The integrated concurrent unit testing framework **ConTestNG Framework** is provided as Java library to facilitate reuse in other Java Applications. The integrated framework invokes JPF for each concurrent unit test and observes the JPF output for exceptions, which indicate potential bugs. Inside the core framework, ConTestNGListener depends on two interfaces: JPFListener and TestNGListener. The JPFListener notifies about thread scheduling, violations such as deadlocks and complete execution history. TestNGListenerAdapter notifies about assertion violations and other uncaught exceptions thrown by TestNG. ConTestNGListener processes results from both interfaces and provides uniform interface to Eclipse plug-in. Since JPF executes in its VM, exceptions thrown by TestNGListenerAdapter are accessible in host VM through the JPF Model Java Interface(MJI). It allows execution of classes under host VM instead of JPF VM through a Peer counterpart class for TestReportListener.

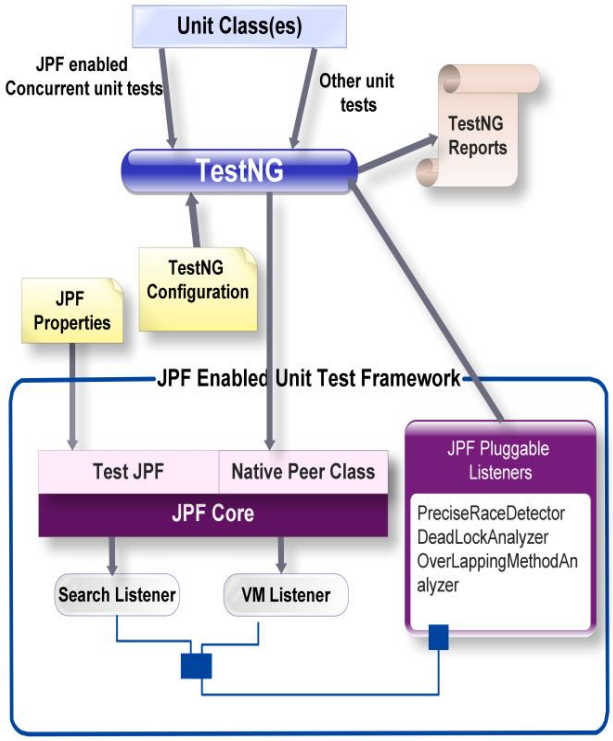

Figure 4 : JPF and TestNG Integration

4.3 Sample Concurrent Unit Test

4.3.1Tested Class:

```
public class RaceConditionWorkQueue {
private LinkedList<String> queue = new LinkedList<String>();
 public void enqueue(String str) {
 synchronized (queue) {
       queue.addLast(str);
       lock.notifyAll();
    }      }
  public int getCurrentWorkPoolSize() {
               return queue.size();      }
  public void work() {
    String current;
    synchronized(queue) {
      if (queue.isEmpty()) {
        try {
          lock.wait();
        } catch (InterruptedException e) {
          assert(true);
        }       }
      current = queue.removeFirst();
    }       System.out.println(current);     }
```

4.3.2 Unit Test Class:

```
public class RaceConditionWorkQueueTest {
            private RaceConditionWorkQueue tested;
            @BeforeMethod
            public void init() {
            tested=new RaceConditionWorkQueue();  }
@Test()
public void testSequentialEnqueue() {
            UUID jobUUID = UUID.randomUUID();
            String jobName = jobUUID.toString();
            tested.enqueue(jobName);
try {
            assertEquals(tested.getCurrentWorkPoolSize(), 1);
} catch (InterruptedException ie) {
            ie.printStackTrace();  }     }
@ConcurrentTest()
@Test(threadPoolSize = 6)
public void testConcurrentEnqueue1() {
            UUID jobUUID = UUID.randomUUID();
            String jobName = jobUUID.toString();
            tested.enqueue(jobName);
try {
            ConUnitTestBarrier.waitForAllThreads();
            assertEquals(tested.getCurrentWorkPoolSize(), 6);
} catch (InterruptedException ie) {
            ie.printStackTrace();
}    }
@ConcurrentTest
@Test(threadPoolSize = 6, dependsOnMethods = { "testConcurrentEnqueue1" } )
public void testConcurrentWork1() {
            tested.work();
try {
            ConUnitTestBarrier.waitForAllThreads();
            assertEquals(tested.getCurrentWorkPoolSize(), 0);
} catch (InterruptedException ie) {
            ie.printStackTrace();
}      }
@ConcurrentTest(threadGroup = "concurrent")
@Test(threadPoolSize = 6)
public void testConcurrentEnqueue2() {
            UUID jobUUID = UUID.randomUUID();
            String jobName = jobUUID.toString();
```

```
        tested.enqueue(jobName);
try {
        ConUnitTestBarrier.waitForAllThreads();
} catch (InterruptedException ie) {
        ie.printStackTrace();
}       }
@ConcurrentTest(threadGroup = "concurrent")
@Test(threadPoolSize = 6)
public void testConcurrentWork2()
   {
        tested.work();
try {
        ConUnitTestBarrier.waitForAllThreads();
} catch (InterruptedException ie) {
        ie.printStackTrace();  } } }
```

4.3.3 Annotations/Helper Classes provided by Framework

- **ConcurrentUnitTest** – to distinguish a concurrent unit test.
- **threadGroup** – can be used to group two similar test cases to be run concurrently.
- **threadPoolSize** – no of threads
- **ConTestNGBarrier** – provides a construct to await for all threads to terminate

Only the tests which are annotated as "ConcurrentUnitTest" are run through JPF and all other tests are run through the sequential TestNG flow. This also allows the developer to retain the existing tests as it is and no code change is required to run them through the framework.

In section 4.3.1, the source code for the RaceConditionWorkQueue class under test is depicted. It is a typical producer/consumer based work queue. It has two methods enqueue and work to submit and assign work accordingly. Conflicting access to the queue are avoided through the use of a synchronized block in enqueue and work functions. And also in section 4.3.1 unit test class for the RaceConditionWorkQueue class is shown. RaceConditionWorkQueue class is instatianiated before each unit test through the Init method() since it is annotated with @BeforeMethod. The first unit test cases testSequentialEnqueue() checks whether enqueue method works fine in a sequential case. The next testcase testConcurrentEnqueue1() checks whether the same method executes correctly if called from multiple threads concurrently. The unit test uses the thread pool size annotation threadPoolSize to specify the number of threads executing the test. The concurrent unit testing frameworks starts the specified number of threads.

The static helper class ConUnitTestBarrier.waitForAllThreads() provides a synchronization construct. It implements a barrier which can be used to ensure that the threads finish executing the concurrent block before verifying the assertions. The barrier is initialized with the number of threads specified in thread pool annotation. An assert is done to check if the current work pool size matches the expected size.

The next testcase testConcurrentWork1(): verifies the work method of RaceConditionWorkQueue in concurrent environment. The *dependsOnMethod* annotation is used to ensure that the workqueue is initialized with the required number of work items before the current testcase is executed.

The thirds set of test cases: testConcurrentEnqueue2 and testConcurrentWork2 use a thread group annotation that can be used to test two or more test methods that can be called concurrently from separate threads. These two tests checks for concurrent issues during parallel execution of enqueue and work through all possible thread interleavings. this simulates real time usage of this class.

4.4.4 Differentiating between Sequential and Concurrent Bugs

The results of the concurrent unit test with different schedulings are compared to each other. There are three basic cases. If the test fulfills the developer's expectation in all possible schedulings, then the test succeeds. If there are schedulings fulfilling the developer's expectations while there are others violating the assertions or triggering run-time errors, then it is obviously a concurrency bug. Finally, if the test does not fulfill the expectations in any of the possible schedulings, then further analysis is needed to determine whether it's a sequential or concurrency bug. The exception thrown during fault-triggering schedulings holds enough information to tell concurrency and ordinary bugs apart in most cases. The exception's type gives details about the type of the failure, its stack trace about the place where the failure has occurred, its message and the causing exception(in case of a chained exception).

1.如果测试在所有可能的调度中都满足了开发人员的期望，那么测试就成功了
2.如果有的调度符合开发人员的预期，而另一些调度违反了断言或者触发了运行时错误，明显是一个并发故障
3.如果所有可能的调度都不符合预期，需要确定是顺序故障还是并发故障。

5. RESULTS & EVALUATION

The proposed framework reliably detects deadlocks, race condition, atomicity violations and order violations. Out of a total of 258 unit tests for one of the platforms, 47 were written as concurrent unit tests. As observed in the results shown in Figure 5, concurrency issues were identified in 15 of the tests.

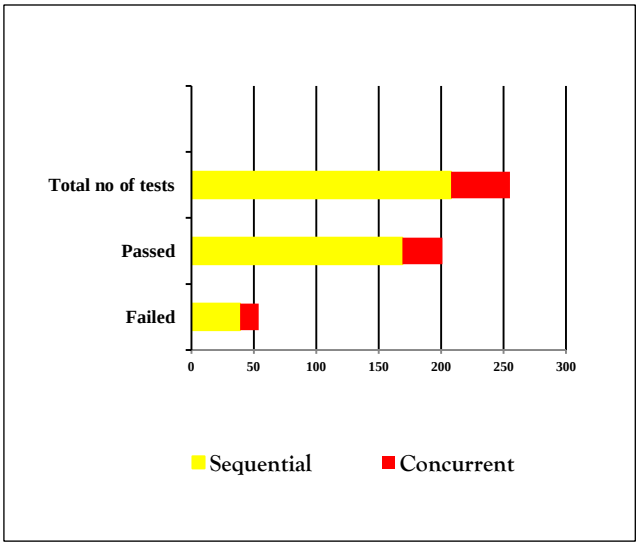

Figure 5 : Results

The existing unit tests do not require code change except for adding annotations. The developer doesn't need to specify a specific sequence of interleaving threads to test the system for

concurrency conditions. All possible thread schedulings and interleavings are automatically explored by JPF. The framework does not modify the original code to add synchronization. Hence it does not interfere with the timing and scheduling of the tested component. This framework helped in identifying issues early and reduced investigation time and effort required for such issues in production.

6. LIMITATIONS

JPF cannot work on native code. JPF configuration is not that extensible and flexible.

7. CONCLUSION

JPF requires a running application and can be run only in later stages of development or testing. The proposed framework solves this issue by integrating JPF with TestNG for concurrent unit testing Adoption of such practices have helped in reducing concurrency related issues in our platforms to a great extent. Issues are being identified early in the development process and this has led to better reliability. Time and effort required for testing and debugging such issues in production has also been reduced dramatically.

The proposed framework can not only be used for concurrent unit testing but also for other unit test cases like boundary condition violations etc, for which JPF already provides extensions. JPF has very active contributions from open source community and the framework can be extended to use such extensions in future.

8. REFERENCES

[1] W.Pugh and N.Ayewah. Unit testing concurrent software. In ASE')7: Proceedings of the twenty-second IEEEE/ACM International Conference on automated software engineering, ACM, pages 513-516, NY,USA, 2007.

[2] B.Long, D.Hoffman, P.Strooper. Tool Support for testing concurrent Java components, IEEE Transactions on Software Engineering,29(6):555-566, 2003.

[3] Brain Goetz, Joshua Bloch, Tim Peierls, Java Concurrency in Practice, Addison Wesley, July 2009.

[4] Doug Lea. Concurrent Programming in Java, Second Edition. Addison –Wesley 2000.

[5] MultithreadedTC.http:/code.google/p//multithreadedTc.com

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

[7] Orit Edelstein, Eitan Farchi, Evgeny Goldin, Yarden Nir, Gil Ratsaby, Shmuel Ur: Framework for testing multi-threaded Java programs. Concurrency and Computation: Practice and Experience 15(3-5): 485-499, 2003 .

[8] Orit Edelstein, Eitan Farchi, Yarden Nir, Gil Ratsaby, and Shmuel Ur Multithreaded java program test generation. IBM Systems Journal, 41(1):111–125, 2002.

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

[10] TestNG unit testing framework http://testng.org The JPF Testing Framework http://JPF.sourceforge.net