

# Testing Java Interrupts and Timed Waits

Luke Wildman   Brad Long   Paul Strooper  
School of Information Technology and Electrical Engineering  
The University of Queensland, 4072, Australia  
email: { luke, brad, pstroop } @itee.uq.edu.au

## Abstract

*Testing concurrent software is difficult due to problems with inherent non-determinism. In previous work, we have presented a method and tool support for the testing of concurrent Java components. In this paper, we extend that work by presenting and discussing techniques for testing Java thread interrupts and timed waits. Testing thread interrupts is important because every Java component that calls `wait` must have code dealing with these interrupts. For a component that uses interrupts and timed waits to provide its basic functionality, the ability to test these features is clearly even more important. We discuss the application of the techniques and tool support to one such component, which is a non-trivial implementation of the readers-writers problem.*

## 1 Introduction

Testing concurrent software is difficult due to problems with inherent non-determinism. When test cases are run multiple times, they may produce different results. This may lead to problems with determining expected outputs and means that coverage criteria for the testing of sequential software may not be sufficient for the testing of concurrent software.

In previous work, we have developed a method and tool support for the testing of concurrent Java components [10]. With the method, the tester first identifies a set of test conditions that will be used to test the component and then constructs a set of test sequences, each consisting of multiple test processes, that will exercise those test conditions. ConAn (Concurrency Analyser) provides tool support for the method.

The method is independent of the test conditions selected. In practice, we have used conditions that cover the loops that typically surround Java `wait` statements zero, one and many times; that vary the num-

ber and types of threads suspended on each monitor queue in the component; and that consider any special component state or parameter values that we want to test. No support for the testing of interrupts or the testing of timed waits was considered in this earlier work. However, since the Java `wait` method can throw `InterruptedException`, this means that all ConAn test scripts presented in [10] do not achieve statement coverage of the code under test, because this exception is never thrown during the testing.

In this paper, simple techniques are presented for the testing of Java interrupts and timed waits. With these techniques, achieving statement coverage becomes straightforward. Moreover, for components where these features are actively used to provide the required functionality of the component, techniques such as this are essential to provide adequate test coverage. One such component, which motivated the work described in this paper, is Doug Lea's `WriterPreferenceReadWriteLock` implementation of the readers-writers problem [9]. This implementation involves three classes and two locks to make the component more efficient. After describing the basic techniques for testing interrupts and timed waits, we show and discuss how they were applied to this more complicated example.

Section 2 presents a simple producer-consumer Java monitor that will be used as a running example and also discusses a ConAn test script for this monitor. Section 3 discusses the testing of interrupts and Section 4 the testing of timed waits. In Section 5, the testing of `WriterPreferenceReadWriteLock` is discussed. Related work is presented in Section 6 and the paper is concluded in Section 7.

## 2 Background

To illustrate the techniques, the simple Java `Buffer` monitor shown in Figure 1 will be used. The monitor implements a finite buffer that may be shared by

multiple producers and consumers. The `put` method is used to add an `Object` to the buffer, and `get` is used to remove an `Object` from the buffer. The `put` and `get` methods are both `synchronized` so that it is guaranteed that only one thread is executing code inside this monitor at a time. Also, when `put` is called and the buffer is full, the calling thread is suspended using a call to `wait`, and similarly a thread that calls `get` is suspended when the buffer is empty. Since a call to `wait` can throw the `InterruptedException`, these calls are surrounded by a try-catch block, which simply ignores the exception in this case. Finally, after the new `Object` has been added to the buffer in `put`, `notifyAll` is called to notify any suspended consumer threads that there now is one more item stored in the buffer. Similarly, `notifyAll` is called in `get` to notify any suspended producer threads that there is now one more space in the buffer.

To test a component with ConAn, the tester performs three steps [10]:

- The tester identifies a set of *test conditions* that will be used to test the component.
- The tester constructs sequences of calls to the component that will exercise the test conditions identified in the first step. Enough test sequences are constructed to cover all test conditions.
- For each test sequence, the tester constructs a set of test processes (Java threads) that will execute the calls as defined in the previous step.

The ConAn (Concurrency Analyser) tool automates the third step in the method.

In previous work [10], we have used test conditions that:

- cover the loops that typically surround Java `wait` statements zero, one and many times;
- vary the number and types of threads suspended on each monitor queue in the component; and
- consider any special component state or parameter values that we want to test.

For the `Buffer` class, 13 test conditions are identified in this way.

These test conditions must then be covered by a set of test sequences, each of which consists of calls to the `put` and `get` operations. Each of these in turn then has to be implemented as a test sequence in a ConAn script, which then automatically generates a test driver with the appropriate test processes in it.

One such ConAn test sequence is shown in Figure 2. A ConAn test sequence is delimited by `#begin` and

```
public class Buffer {
    Object[] buf;
    int in = 0;
    int out= 0;
    int count= 0;
    int size;

    public BufferImpl(int size) {
        this.size = size;
        buf = new Object[size];
    }

    public synchronized void put(Object o) {
        while (count==size) {
            try { wait(); }
            catch (InterruptedException e) {}
        }
        buf[in] = o;
        ++count;
        in=(in+1) % size;
        notifyAll();
    }

    public synchronized Object get() {
        while (count==0) {
            try { wait(); }
            catch (InterruptedException e) {}
        }
        Object o =buf[out];
        --count;
        out=(out+1) % size;
        notifyAll();
        return (o);
    }
}
```

**Figure 1. A Java implementation of a bounded buffer**

```
#begin
#tick
#thread
    #valueCheck (String)b.get() # "a" #end
    #valueCheck time() # 2 #end
#end
#end
#tick
#thread
    #excMonitor b.put("a"); #end
    #valueCheck time() # 2 #end
#end
#end
#end
```

**Figure 2. A ConAn test sequence for Buffer**

**#end**. Each test sequence consists of a number of *tick blocks*, delimited by **#tick** and **#end**. Each tick block represents a unit of time. The progression of time is modelled by an external clock, which is controlled by ConAn. Within each tick block, there are a number (often one) of *thread blocks*, delimited by **#thread** and **#end**. ConAn creates a thread for each thread block *t*. This thread executes the Java code that appears within *t* when the clock reaches the time associated with the tick block in which *t* appears. To support the checking of values and exceptions, we have integrated ConAn with the Roast unit testing tool [4]. Arbitrary Java code, as well as Roast *test templates*, can appear within a thread block.

The test sequence in Figure 2 checks that a call to **get** will suspend when the buffer is empty, and that a subsequent call to **put** should wake up the suspended thread and return the item that has just been stored in the buffer. For the first tick block in the test sequence, ConAn creates a thread that calls **get** on the buffer object **b**, which has been declared in the ConAn script and is instantiated by ConAn for each test sequence. The call to **get** should suspend, because the buffer is empty. The call is contained in a Roast value-checking test template that checks that the return value of the call is "a" when the call returns. After this has been checked, the thread will execute another value-checking test template that checks that the ConAn clock is at time 2 (this represents the fact that we expect the call to **get** to complete at time 2, when the call to **put** is made).

For the second tick block, ConAn again creates a thread that calls **put**, but because this call appears in the second tick block, ConAn will delay this call until the ConAn clock reaches time 2. The call is placed inside an exception-monitoring test template to ensure that no exception is thrown when this call is executed. After the call to **put** completes, the ConAn clock is checked again to make sure that this call also completes at time 2.

The entire ConAn script for this example contains 6 such test sequences to cover the 13 test conditions identified [10]. ConAn generates a Java test driver from this script. The test driver code sets up the clock, instantiates the threads for each test sequence, generates calls to control the suspension of threads until the appropriate time, and manages the passing of time. ConAn also detects if a thread is suspended indefinitely at the end of a test sequence. If this happens, it reports an error, terminates the thread, and continues with the next test sequence. On completion of the test script, the number of test cases and errors are reported.

When the test script for the **Buffer** class is exe-

cuted, we do not achieve 100% statement coverage of the component under test. In particular, we do not execute the statements

```
catch (InterruptedException e) {}
```

in either **put** or **get** because threads are never interrupted during the testing. Note that removing the try-catch block and replacing it with a **throws InterruptedException** on the method declaration does not really solve this problem, because this code would still not be executed. Such a change would also change the behaviour of the component: currently, if a thread that has called **put** or **get** is interrupted while or before it has called **wait**, then this interrupt will be ignored; with the changed code, the interrupt would be propagated to the calling code.

### 3 Testing Interrupts

A detailed account of Java Interrupts appears in [9]. Threads may be interrupted by a call to the method **interrupt**. Because threads may be interrupted while they are inside critical sections (synchronised blocks), it is not reasonable to abruptly stop the execution of a thread or direct it to a handler. To allow the programmer to control the behaviour of threads, each **Thread** has an associated boolean *interrupt status*, which can be checked by calling the method **interrupted**. In most cases, a thread interruption simply changes the interrupt status to true. Threads need to call **interrupted** to see if they have been interrupted. This allows a thread to respond to the interrupt in any way desired, and in particular, the thread may complete execution of critical sections. To facilitate reasonable responsiveness to interrupts, threads should regularly check whether they have been interrupted.

Threads that have called **wait**, **sleep**, or **join** automatically throw **InterruptedException** if the interrupt status is true upon execution of the call, or if **interrupt** is called while they are suspended. Hence the requirement that calls to these methods be placed in blocks that indicate that they may throw this exception or catch it. Raising an exception at a known point allows suspended threads to respond to interrupts as well as tidy up before cancelling the thread (or some other response). The interrupt status is cleared when the exception is raised and as a side effect of calling **interrupted**.

Note that threads suspended on **synchronization** locks do not respond to interrupts. So, for example, threads that are deadlocked in a deadly embrace wait-

ing for each other to release locks on monitors cannot be released by using interrupts.

### 3.1 Possible faults

As described above, an incorrect response to an interrupt may potentially corrupt shared data. In addition, as a waiting thread is woken automatically (and removed from the wait set) by thread interruption, it is possible that a notification intended for that thread is unintentionally lost or directed to the wrong thread. It may thus be necessary for the interrupted thread to send extra notifications to avoid a dormant thread that is never woken up.

The issue of a lost notification can be illustrated with the `Buffer` class in Figure 1. Suppose the buffer is empty and a consumer thread  $t$  is suspended on the call to `wait` inside `get`. If another thread interrupts  $t$  just before a third producer thread that has called `put` calls `notifyAll`, then the `notifyAll` will not wake up  $t$ . Of course, in this case this does not lead to a problem, because in both the cases in which thread  $t$  is interrupted or notified, it will recheck if there are any items in the buffer and exit the loop surrounding the `wait`.

### 3.2 Technique for testing interrupts

For each method that is supposed to deal with interrupts, we want to check the behaviour of the interrupt-handling code. The obvious examples are calls to the `interrupted` method and calls to `wait` dealing with `InterruptedException`. In general, we include test cases for such code that:

1. Interrupts the calling thread  $t$  when it is suspended after having called `wait` in the method under test.
2. Interrupts the calling thread  $t$  before the method under test is called by  $t$ .

The checking we do in each of these cases depends on the required behaviour in response to an interrupt.

For example, for the bounded buffer component in Figure 1, it was decided to ignore the `InterruptedException` when it is thrown. Following the technique outlined above, there are two test cases that we would use to test this for both `get` and `put`.

1. Force one producer/consumer thread to call `wait` inside `put/get` and then interrupt the waiting thread. Check that the interrupted thread goes back to a waiting state after it has been interrupted.

```
#begin
#tick
#thread <t1>
#valueCheck (String)b.get() # "a" #end
#valueCheck time() # 3 #end
#end
#end
#tick
#thread
#excMonitor thread("t1").interrupt(); #end
#valueCheck time() # 2 #end
#end
#end
#tick
#thread
#excMonitor b.put("a"); #end
#valueCheck time() # 3 #end
#end
#end
#end
```

Figure 3. A ConAn test sequence for testing an interrupt in `get`

2. Create a producer/consumer thread and interrupt it before it calls `put/get`. Again check that the interrupt has no effect.

The first of these is implemented in the ConAn test sequence shown in Figure 3. To support the technique, we have extended ConAn to provide the ability to name and access threads in code associated with other threads<sup>1</sup>. For example, the thread that executes at time 1 in the test case in Figure 3 is named `t1` and this name is used in the thread block at time 2 to interrupt that thread. Note that we check that the completion time of the first call to `get` is time 3, which is not until a call to `put` has been made. The test cases for interrupting the thread before it has called `get` and the test cases for checking the interrupt in `put` are similar.

If the try-catch block was removed and replaced by `throws InterruptedException` on the method declaration, the first two lines in the above test sequence would have to be changed to

```
#excMonitor b.get(); #
new InterruptedException() #end
#valueCheck time() # 2 #end
```

where the call to `get` now completes at time 2 by throwing `InterruptedException`. As a result, the code for

<sup>1</sup>The naming facility already existed in ConAn, but in previous versions, this name could not be used to reference the thread in other thread blocks.

the third tick time in the test sequence is no longer needed.

### 3.3 Discussion

If we add the above four test cases to the existing ConAn test suite, we achieve 100% statement coverage of the code under test. Achieving such coverage is often easy to achieve, and while we view it as a “necessary” condition for adequate testing, it is a relatively weak criterion and clearly not sufficient by itself. It is much more prudent to test that all the requirements have been implemented, including any requirements related to interrupt handling. Of course, what these requirements are, varies from one component to the next.

The technique we have presented above only tests interrupts after a thread is suspended on a call to `wait`, or before the thread has called the method under test. There are other cases that might be of interest as well. For example, *after* the thread has called the method under test, but *before* it has called `wait`. However, this is very hard to test with a tool such as ConAn, because it would involve interrupting a thread at exactly the right point in time, which is something we do not have control over. In testing terms, we do not have the required controllability of the code under test. As such, we have not included cases such as this in our technique and suggest using other verification techniques such as code inspection if such checking is deemed necessary.

In this paper, we focus on testing interrupts in relation to calls to `wait`. As noted in Section 3, interrupts also affect calls to `join` and `sleep`, but we have not encountered these in the components we have tested so far and as such they are not discussed in detail in this paper.

## 4 Testing timed waits

Timed waits are used to control the amount of time that a thread remains suspended by a `wait` statement. The time parameter gives a lower bound on the maximum amount of time that the thread should remain waiting. Should the thread not be notified in the given period, the thread will be removed from the wait set and become eligible to receive the lock. While it is guaranteed that the thread will wait at least as long as the parameter dictates, there is no corresponding guarantee on the upper bound. Note that `wait(0)` corresponds to a normal (untimed) wait.

Figure 4 shows a variation of the `put` method that waits for a given time `msecs` if the buffer is full. The modified call returns `true` if the `Object` is added to the buffer, and `false` otherwise. The new code checks

```
public synchronized
boolean put(Object o, long msecs) {
    if (count==size) {
        long waitTime = msecs;
        long start = System.currentTimeMillis();
        for (;;) {
            if (waitTime <= 0)
                return false;
            else {
                try { wait(waitTime); }
                catch (InterruptedException e) {}
                if (count < size)
                    break;
                else
                    waitTime = msecs
                        - (System.currentTimeMillis()
                          - start);
            }
        }
    }
    buf[in] = o;
    ++count;
    in=(in+1) % size;
    notifyAll();
    return true;
}
```

Figure 4. A timed version of `put`.

that time bound has not been exceeded following a (unnecessary) notification or timeout before waiting again. In the case of a timeout value of zero, the method will not wait at all if the buffer is full. Note that the conventional embedding of a `wait` within a while loop is more complex for timed waits. An alternative scheme is discussed in [9].

### 4.1 Possible faults

An incorrect parameter to a timed wait may result in the thread waiting either too long or not long enough. In addition, timed waits suffer all the possible faults of a normal `wait` statement (incorrect wait condition, etc. [11]). As timed waits allow an uncontrolled exit from the wait state, they are also vulnerable to missed notifications in the same way as interrupted waits.

### 4.2 Technique for testing timed waits

For each method that is supposed to deal with timeouts, we want to check the behaviour of the timeout-handling code. We include test cases for such code that:

```

#ticktime 200
#begin
#tick //1
#thread <t1>
#valueCheck (Boolean)b.put("a",100) # true #end
#valueCheck time() # 1 #end
#end
#end
#tick //2
#thread <t2>
#valueCheck (Boolean)b.put("b",100) # false #end
#valueCheck time() # 2 #end
#end
#end
#tick //3
#thread <t3>
#valueCheck (String)b.get() # "a" #end
#valueCheck time() # 3 #end
#end
#end
#end

```

**Figure 5. A ConAn test sequence for testing a timed wait in put**

- Checks the behaviour of a call to `wait` that times out and one that does not time out.
- Checks various interesting values for the time parameter, provided that value can be controlled by the tester.

The checking we do in each of these cases depends on the required behaviour in response to a timeout. For example, in the timed `put` method in Figure 4, the required behaviour in response to a timeout is to abort the attempt to add the item to the buffer and to indicate the timeout by returning `false`.

There are four test cases that we use to test the timed `put` method.

1. Cause a timeout on the only waiting thread.
2. Cause a timeout when there are many waiting threads. This is necessary to test the update to `waitTime` after the call to `wait`.
3. Call the timed method with a timeout value of 0.
4. Call the timed method with a (large) value that does not cause a timeout.

The first of these is implemented in the ConAn test sequence shown in Figure 5. This sequence assumes that the maximum size of the buffer is one item.

In the example, the first `put` succeeds immediately and returns `true`. The second call to `put` must wait. It times out after 100 milliseconds and returns `false`. ConAn allows the timing of the ticks to be controlled with the `#ticktime` command. By setting this to 200 milliseconds the timed out thread (normally) terminates in tick 2. However the Java language specification only guarantees a lower bound on the maximum wait time for a timed wait. Therefore it is not possible to guarantee that the thread will timeout in tick 2, and in fact, it may not timeout at all if it runs over into the following tick. In practice, this does not present a real problem. In the examples we have looked at, we have been able to cause threads to time out reliably by choosing a large enough value of `ticktime` compared to the wait time.

### 4.3 Discussion

Full coverage of the time related parts of the code is achieved with the test cases suggested. Once again, code coverage is a “necessary” condition but is not sufficient by itself.

The testing strategy is not a “real-time” testing strategy because Java provides no real-time guarantees. However, ConAn enables suitable test sequences to be devised in the cases we have seen.

## 5 Case Study

### 5.1 Readers-writers example

We now discuss testing of the entire `WriterPreferenceReadWriteLock` monitor from Doug Lea’s concurrency utilities package included as part of the upcoming release of JSR166 [12]. The ConAn tool and test scripts devised as part of this case study are available at <http://www.itee.uq.edu.au/~testcon>.

The class `WriterPreferenceReadWriteLock` implements a two-lock version of the readers-writers monitor: reader and writer threads communicate via a shared resource, simultaneous reads are allowed but in order to prevent interference, writing requires exclusive access. Having separate locks for readers and writers minimises the number of waiting threads that will be notified unnecessarily (i.e. fail the monitor entry condition after notification). In this version of the monitor, waiting writer threads are given preference over readers. The size of the monitor precludes showing the complete code (309 LOC). The class fragment `ReaderLock` given in Figure 6 demonstrates its complexity and the concepts involved.

```

protected class ReaderLock extends Signaller
implements Sync {
...
public void release() {
    Signaller s = endRead();
    if (s != null) s.signalWaiters();
}

synchronized void signalWaiters() {
    ReaderLock.this.notifyAll(); }

public boolean attempt(long msecs)
throws InterruptedException {
    if (Thread.interrupted())
        throw new InterruptedException();
    InterruptedException ie = null;
    synchronized(this) {
        if (msecs <= 0)
            return startRead();
        else if (startReadFromNewReader())
            return true;
        else {
            long waitTime = msecs;
            long start = System.currentTimeMillis();
            for (;;) {
                try {
                    ReaderLock.this.wait(waitTime);
                }
                catch(InterruptedException ex){
                    cancelledWaitingReader();
                    ie = ex;
                    break;
                }
                if (startReadFromWaitingReader())
                    return true;
                else {
                    waitTime = msecs
                        - (System.currentTimeMillis()
                            - start);
                    if (waitTime <= 0) {
                        cancelledWaitingReader();
                        break;
                    }
                }
            }
        }
    }
}
// safeguard on interrupt or timeout:
writerLock_.signalWaiters();
if (ie != null) throw ie;
else return false; // timed out
}
...
}

```

**Figure 6. A fragment of Doug Lea's Writer-PreferenceReadWriteLock monitor**

The monitor `WriterPreferenceReadWriteLock` contains 14 methods and two nested subclasses, `ReaderLock` and `WriterLock`, each containing 4 methods. The `ReaderLock` provides methods `attempt`, `acquire` (not shown), and `release` to reader threads and the `WriterLock` provides similar methods to writer threads.

Method `attempt(msecs)` implements a timed attempt to get read access to a resource. The method returns `true` if read access may be acquired before ( $system.currentTimeMillis + msecs$ ) and `false` otherwise. If the parameter `msecs` is zero or negative the method attempts to begin reading immediately and returns `false` if this is not possible. Method `acquire` provides untimed access to the resource. Methods `release` and `signalWaiters` are used to implement the release part of the readers-writers protocol.

The parent class defines a number of methods that are used to implement the writer-preference policy. For instance, the various `startRead` methods are used to test whether reader threads may start reading when entering the monitor in various ways; e.g., for the first time or after waiting.

The `attempt` method checks for interruption before entering the synchronized block to avoid unnecessary synchronization. In the case of interruption or timeout while the thread waits to acquire read access, the `attempt` method must tidy up by decrementing the number of waiting readers (via `cancelledWaitingReader`) and, to safeguard against the possibility that a notification goes astray, an extra notification is issued to the writer lock. The writer lock is notified because it is possible, in the case that this is the only waiting reader thread, that a notification sent to the interrupted reader thread was lost, and that one should now be directed to a writer thread waiting on the interrupted thread.

## 5.2 Testing a readers-writers monitor

The application of our test method for testing concurrent monitors without consideration of interrupted or timed waits is now described.

Loop coverage is achieved if we consider 3 test conditions: (C1) a reader thread starts reading immediately, (C2) a reader thread starts reading following one iteration of the `for` loop, (C3) a reader thread starts reading after multiple iterations of the `for` loop (i.e after being notified and failing the start read condition). Matching sets of conditions are produced for the other methods `ReaderLock.acquire` (C4-C6), `WriterLock.attempt(msecs)` (C7-C9), and `WriterLock.acquire` (C10-C12).

In addition, we produce test conditions that vary the number and type of thread suspended in each call to `notify/notifyAll` in `signalWaiters`. Calls to `notify/notifyAll` arise from calls to `release` that cause the `ReaderLock` or `WriterLock` to be notified. Eighteen other conditions for both the `attempt` and `acquire` methods are derived this way. (E.g. “no readers or writers waiting”, and “many writers and many readers waiting on a reader locking”.)

Finally, the method considers any special state and parameter conditions. We test the ability of the monitor to allow many readers to run simultaneously here. Other special conditions are all related to the handling of interrupts and timed waits and are treated separately in Sections 5.3 and 5.4.

### 5.2.1 Test sequences

Sequences of calls to the monitor methods are constructed to cover the 30 conditions identified above. It is frequently the case that test sequences actually test a number of conditions, however, we do not attempt to minimize the number of test sequences.

Test sequences for conditions C1 and C2 are produced following the procedure outlined in Section 2. However, condition C3 is very difficult to test because in normal operation of the monitor, threads are notified only when their monitor entry condition is satisfied (hence the added efficiency of the two-lock implementation). Execution of condition C3 relies either on the monitor entry condition becoming false due to a race condition, or due to the production of a redundant notification when the monitor entry condition is false. This turns out to be possible in the case of interruption, for instance an interrupted writer thread unconditionally notifies the readers in case any reader threads are waiting on it to terminate. We will return to this condition in Section 5.3. Similar reasoning applies for the other methods. In total, there are 4 conditions of the same type as C3 that are difficult to test without consideration of the behaviour of interrupts.

Construction of test sequences for the remaining conditions relies on knowledge of the “writer priority” policy to cause reader and writer threads to wait. For example, “many readers waiting” can be achieved by first giving access to a writer thread by calling `WriterLock.acquire` and then causing reader threads to wait by several calls to `ReaderLock.attempt`.

Following the procedure outlined above, 20 test sequences were produced in total. Execution of the test sequences resulted in 232 calls to the monitor and no errors were reported.

### 5.2.2 Coverage of simple test cases

The conditional, statement and method coverage of the test set was measured using the Clover [3] coverage tool. Coverage data for the parent class `WriterPreferenceReadWriteLock` and its nested subclasses is presented in Table 1. Although 100% method

Class	Condition	Statement	Method
<code>WriterLock</code>	54.5%	55.1%	100%
<code>ReaderLock</code>	54.5%	58.7%	100%
Parent	88.9%	94.7%	84.6%

**Table 1. Clover coverage - simple test cases.**

coverage is achieved in the Lock classes, very poor statement and condition coverage is achieved. This lack of statement coverage is due to the failure to execute any of the interrupt or timeout code as expected, however the poor condition coverage is primarily due to the fact that the monitor entry condition is always true when a thread is woken. For instance, in the method `attempt` presented above, the call to `startReadFromWaitingReader` always returns true. This also accounts for the poor condition coverage in the parent class `WriterPreferenceReadWriteLock`. The incompleteness in the method and statement coverage of the parent class is entirely accounted for by the fact that methods `cancelledWaitingReader` and `cancelledWaitingWriter` are never executed because interrupts and timeouts are not tested.

### 5.3 Additional test cases for interrupt

We now consider how to test the interrupt related code in the case study following the method described in Section 3.

To achieve interrupt and exception coverage we use the following conditions (which apply to the example `attempt` above): `Thread.interrupted` is true before the check for interruption prior to entering the synchronized block, interrupting a thread while it waits, and finally, the call to `signalWaiters` and the throw of `InterruptedException`. Matching sets of conditions are produced for the other `acquire` and `attempt` methods.

To test correct use of `notifyAll/notify` to tidy up the monitor state, we identify 24 conditions in which a waiting thread is interrupted with zero or more other reader and writer threads waiting (for both `attempt` and `acquire` methods).

### 5.3.1 Test sequences

Test sequences for these conditions are constructed as described in Section 3.

Note that the use of interrupts allows us to test conditions *C3*, *C6*, *C9*, and *C12* which cause multiple iterations of the wait loop. These are difficult to test otherwise because of the unlikely event of a thread being woken with a false entry condition. However, as interrupts cause exceptional behavior in the monitor, notifications can be sent at any time.

Twenty-eight new test sequences were constructed to cover the 32 conditions described above. Execution of the test sequences resulted in an extra 384 calls to the monitor and no errors were reported.

### 5.3.2 Coverage with interrupts

Coverage data is presented in Table 2. This is the

Class	Condition	Statement	Method
WriterLock	86.4%	89.8%	100%
ReaderLock	86.4%	91.3%	100%
Parent	100%	100%	100%

**Table 2. Clover coverage - interrupt test cases.**

combined coverage data of the simple test cases and the interrupt test cases. The combined test cases achieve coverage of all normal and interrupt based code. The gaps in coverage are entirely due to the failure to test the time-out code in the `attempt` methods.

## 5.4 Additional test cases for timed wait

Complete coverage of the `attempt` methods from the `ReaderLock` and `WriterLock` classes is possible by using the technique outlined in Section 4.

The following conditions have been chosen to test the checks for time-outs in `ReaderLock` and `WriterLock`: parameter `msecs` less than or equal to 0 and reader may start immediately, parameter `msecs` less than or equal to 0 and reader may *not* start immediately, a timeout occurs, and a timeout never occurs. The final condition in which a timeout never occurs has been considered in the previous test sections so we need not consider it further.

The remaining criteria are to do with correct thread cancellation after a timeout. This results in 18 new test conditions.

### 5.4.1 Test sequences

Test sequences covering the 18 conditions described above involve calls to `attempt` with values of parameter `msecs` chosen to cause a timeout in the desired condition. 15 new test sequences are constructed.

### 5.4.2 Coverage with timed waits

Execution of all of the above test sequences achieves 100% coverage of statements, conditionals, and methods of the `WriterPreferenceReadWriteLock` class.

## 6 Related Work

Deterministic testing of concurrent programs requires forced execution of the program according to an input test sequence. Brinch Hansen [1] presents a method for testing Concurrent Pascal monitors. He separates the construction from the implementation of test cases, and makes the analysis of a concurrent program similar to the analysis of a sequential program. ConAn [10], the tool adopted in this paper, provides support for an approach to testing concurrent components that extends Brinch Hansen’s approach.

Our approach to testing is largely requirements based, but we also focus on particular language constructs (e.g. `wait/notify/interrupt`) in order to ensure code coverage. Similarly, Carver and Tai [2] use a constraint-based approach to testing concurrent programs, which involves deriving a set of validity constraints from a specification of the program, performing non-deterministic testing, collecting the results to determine coverage and validity, generating additional test sequences for paths that were not covered, and performing deterministic testing for those test sequences. However, these methods require a specification, are hard to apply in practice due to a lack of tool support, and require modification or transformation of the code under test. Our approach has none of these shortcomings.

A limitation of our approach is that it is difficult to test race conditions. Some dynamic analysis tools (RaceFinder [6], ConTest [5]) attempt to increase the chance of failure detection by transforming the code so that it should behave in the same way but is more likely to exhibit concurrent bugs such as race conditions and deadlocks. For example, ConTest is a tool whose main aim is to find data races, deadlocks and other intermittent bugs in multithreaded Java programs. ConTest instruments the bytecode of the application with heuristically controlled conditional `sleep` and `yield` instructions.

The main contributions of this paper are testing strategies for Java interrupts and timed waits. As far as we are aware there have been no other publications in the area of interrupt testing. Sinha and Harrold [13] discuss the effect of exception-handling constructs on data and control flow analyses, and discuss how this affects the structural testing of Java programs with exceptions. Also, in work on the detection and masking of nonatomic exception blocks [7], that is, where an exception leaves an object in an inconsistent state, they describe gains in both line coverage and condition coverage as was reported here. Timed waits are related to the area of real-time testing [8], however, as timed waits do not guarantee real-time performance, a different approach is needed. As far as we are aware, no other testing strategies have addressed timed waits in particular.

## 7 Concluding remarks

We have presented two simple techniques for the testing of Java interrupts and timed waits, and shown how they can be applied using the ConAn testing tool. We have discussed the application of these techniques to Doug Lea's `WriterPreferenceReadWriteLock` component, which is a non-trivial implementation of the readers-writers problem. The case study has shown that the techniques are adequate to achieve at least statement coverage of the code under test. It has also shown that the techniques can be used to assist with the testing of code that does not deal with interrupts, because it allows us to test conditions in the code under test that would be hard to test otherwise. In other words, the use of interrupts has increased the controllability of the code under test.

In this paper, we have focused on testing interrupts in relation to calls to `wait`. We believe that similar techniques could be used to test interrupt-related code dealing with calls to `sleep` and `join`, but have not tried this on any examples.

Currently the technique does not support the testing of interrupts at arbitrary points in the code under test. We suggest using other verification techniques such as code inspection if this is deemed necessary for such cases. Another possibility is to investigate code instrumentation or modification of the JVM to provide greater controllability over the locations at which the code under test can be interrupted.

## Acknowledgements

This research is funded by an Australian Research Council Discovery grant, DP0343877: *Practical Tools and Tech-*

*niques for the Testing of Concurrent Software Components.*

## References

- [1] P. Brinch Hansen. Reproducible testing of monitors. *Software – Practice and Experience*, 8:721–729, 1978.
- [2] R.H. Carver and K-C. Tai. Use of sequencing constraints for specification-based testing of concurrent programs. *IEEE Transactions on Software Engineering*, 24(6):471–490, 1998.
- [3] Cortex. Clover: a coverage tool for Java. <http://www.thecortex.net/clover/index.html>, June 2004.
- [4] N. Daley, D. Hoffman, and P. Strooper. A framework for table driven testing of Java classes. *Software- Practice and Experience*, 32(5):465–493, 2002.
- [5] O. Edelstein, E. Farchi, Y. Nir, G. Ratsaby, and S. Ur. Multithreaded Java program test generation. *IBM Systems Journal*, 41(1):111–125, 2002.
- [6] Y. Eytani, E. Farchi, and Y. Ben-Asher. Heuristics for finding concurrent bugs. In *Proc. of the 17th Int. Parallel and Distributed Processing Symposium (IPDPS 2003) – 1st Int. Workshop on Parallel and Distributed Systems: Testing and Debugging*. IEEE Computer Society, 2003.
- [7] C. Fetzer, P. Felber, and K. Högstedt. Automatic Detection and Masking of Nonatomic Exception Handling. *IEEE Transactions of Software Engineering*, 30(8):547–560, August 2004.
- [8] Anders Hessel, Kim G. Larsen, Brian Nielsen, Paul Pettersson, and Arne Skou. Time-optimal real-time test case generation using UPPAAL. In A. Petrenko and A. Ulrich, editors, *Proc. of 3rd Int. Workshop on Formal Approaches to Testing of Software*, volume 2931 of *LNCS*, pages 114–130. Springer-Verlag, 2003.
- [9] D. Lea. *Concurrent Programming in Java, Design Principals and Patterns*. Addison Wesley, Second edition, 2000.
- [10] B. Long, D. Hoffman, and P. Strooper. Tool support for testing concurrent Java components. *IEEE Transactions on Software Engineering*, 29(6):555–566, June 2003.
- [11] B. Long, P. Strooper, and L. Wildman. A method for verifying concurrent Java components based on an analysis of concurrency failures. *Concurrency and Computation – Practice and Experience*. Submitted for publication, 2004.
- [12] Java Community Process. Java Specification Request 166: Concurrency utilities. <http://www.jcp.org/en/jsr/detail?id=166>, March 2004.
- [13] S. Sinha and M.J. Harrold. Analysis and testing of programs with exception handling constructs. *IEEE Transactions on Software Engineering*, 26(9):849–871, 2000.