Testing Java Monitors through Deterministic Execution

Craig Harvey* and Paul Strooper

School of Computer Science and Elec. Eng.,
Software Verification Research Centre,
The University of Queensland,
Brisbane, Qld. 4072, Australia.
email: charvey@iprg.nokia.com, pstroop@csee.uq.edu.au

Abstract

Java is a popular, modern programming language that supports monitors. However, monitor implementations, like other concurrent programs, are hard to test due to the inherent non-determinism. This paper presents a method for testing Java monitors, which extends the work of Brinch Hansen on testing Concurrent Pascal monitors.

A monitor is tested by executing a concurrent program in which the processes are synchronised by a clock to make the sequence of interactions deterministic and reproducible. The method is extended to account for the differences between Concurrent Pascal monitors and Java monitors, and to provide additional coverage of the implementation under test. Tool support and documentation in the form of a test plan are also provided. The method is illustrated in detail on an asymmetric producer-consumer monitor, which is the same example that was used to illustrate the original method. The application of the method to the readers and writers problem is also discussed.

1 Introduction

Java supports monitors, which encapsulate data that can only be observed and modified by monitor access procedures [13]. Only one access procedure may be active inside a monitor at a time, and a monitor access procedure thus has mutually exclusive access to the data variables encapsulated in the monitor.

The testing of concurrent programs in general, and the testing of monitors in particular, is difficult due to the inherent non-determinism in these programs. For example, if we run a concurrent program twice with the same test input, it is not guaranteed to return the same result both times. This is due to the fact that it is difficult to control the event orderings inside concurrent programs, and hence it is difficult to automate the checking of test outputs.

In this paper, we extend a method for testing monitors proposed by Brinch Hansen [2]. The original method consists of four steps:

1. For each monitor operation, the tester identifies a set of preconditions that will cause each branch (such as those occurring in an if-then-else) of the operation to be executed at least once.

2. The tester constructs a sequence of monitor calls that will exercise each operation under each of its preconditions.

3. The tester constructs a set of test processes that will interact exactly as defined above. These processes are scheduled by means of a clock used for testing only.

4. The test program is executed and its output is compared with the predicted output.

By using an external clock to synchronise the calls to the monitor, we can obtain deterministic behaviour without changing the code under test and thus guarantee that we exercise the preconditions we want to.

However, the original method was devised for monitors implemented in Concurrent Pascal, and due to a number of differences between Concurrent Pascal monitors and Java monitors, the method must be enhanced before it can be applied effectively to Java monitors. In particular, Java monitors do not provide *condition variables* and do not implement the *immediate resumption requirement* [1]. As a result, it is no longer sufficient to merely aim for branch coverage in the first step of the method. Instead we should consider loop coverage, as well as the number and type of processes suspended inside the monitor. In addition, we extend

*Current address: Nokia IPRG, PO Box 4093, Gold Coast, Qld. 4230, Australia

0-7695-1254-2/01 $10.00 © 2001 IEEE

the method by considering interesting state and parameter values, and we provide documentation in the form of a test plan and tool support by incorporating the method into the Roast tool for testing Java classes [6, 10, 9].

We review related work in Section 2. In Section 3, we introduce the Java equivalent of the asymmetric producer-consumer monitor used by Brinch Hansen [2] to illustrate the original method. We describe the enhanced method in detail in Section 4 and apply it to the asymmetric producer-consumer monitor. In Section 5, we discuss the application of the enhanced method to the readers and writers problem.

2 Related Work

Several strategies for the testing of concurrent programs have been proposed in the literature. Static analysis involves the analysis of a program without requiring test execution. Several graphical notations for representing the behaviour of concurrent programs have been proposed [16, 12, 17, 14, 11]. The resulting graphs are then analysed to generate suitable test cases, to generate suitable synchronisation sequences for testing, or to verify properties of the program. However, these techniques all suffer from the *state explosion problem*: even for simple concurrent programs, the resulting graphs are large and complex. In many cases, this problem is compounded by a lack of tool support.

A number of authors [3, 15, 5] have proposed techniques for "replaying" concurrent computations. While helpful, such tools do nothing to achieve adequate test coverage. Carver and Tai [4] 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. This method requires a specification and is hard to apply in practice due to a lack of tool support.

More recently, model checking has been used to automatically test interactive programs written in a constraint-based language [7]. The method uses an algorithm to systematically generate all possible behaviours of such a program, and these behaviours are then monitored and checked against user-specified safety properties.

Very few practical proposals have been made for the generation of test data and the execution of this test data. Our work builds on the work of Brinch Hansen [2], who presents a method for testing Concurrent Pascal monitors. He separates the construction and implementation of test cases, and makes the analysis of a concurrent program similar to the analysis of a sequential program.

3 Producer-Consumer Monitor

The `producerConsumer` class shown in Figure 1 implements an asymmetric Producer-Consumer monitor, which is the Java equivalent of the Concurrent-Pascal version described in [2]. The `send` method places a string of characters into the buffer and the `receive` method retrieves the string from the buffer, one character at a time.

The state of the monitor is maintained through three variables: `contents` is a string that is used to store the string of characters, `curPos` represents the number of characters in the string that still have to be received, and `totalLength` represents the length of `contents`.

The `synchronized` keyword in the declaration of the `send` and `receive` methods guarantees that these methods are executed under mutual exclusion (i.e. only one thread can be active inside one of these methods at any time). If any threads attempt to execute a synchronised method in a class while there is another thread active in the same class, this thread will suspend on a queue.

To block a consumer thread when there are no characters in the buffer and a producer thread when there is still part of a string in the buffer, the `wait` operation is used. It suspends the thread that executed the call and releases the synchronisation lock on the monitor. The `wait` calls are placed inside a `try-catch` block to trap any thread interruption exceptions that may occur. The `notifyAll` operation wakes up all threads that are waiting on the queue for this monitor, although only one thread at a time will be allowed to access the monitor.

There are two main differences between the implementation of monitors in Java and the more traditional monitors used by Brinch-Hansen in Concurrent Pascal. First, in Concurrent Pascal, there are condition variables on which processes suspend so that, for example, producers and consumers typically are suspended on different queues. In Java, there is only a single anonymous condition variable per class (strictly speaking, per object, but for brevity we only consider the simpler case here), which means that producers and consumers are suspended on the same queue. Second, in Concurrent Pascal, the *immediate resumption requirement* guarantees that, when a process signals on a condition variable, the process that has been suspended the longest on a condition variable is the one that gains control of the synchronisation lock. No such guarantee exists in Java monitors.

This means that there are two changes between Brinch-Hansen's implementation of the asynchronous producer-consumer and the version above. First, instead of using an if-condition to check if the current thread should suspend, the Java implementation uses a while-condition. Second, instead of waking up a single thread at the end of each method, all threads are notified. These differences are not

```
class producerConsumer {
   String contents;
   int curPos = 0;
   int totalLength;

   // receive a single character
   public synchronized char receive() {
      char y;
      // wait if no character is available
      while (curPos == 0) {
         try{
            wait();
         }
         catch (InterruptedException e){}
      }
      // retrieve character
      y = contents.charAt(totalLength - curPos);
      curPos = curPos - 1;
      // notify any other blocked send/receive calls
      notifyAll();
      return y;
   }
   // send a string of characters
   public synchronized void send(String x) {
      // wait if there are still characters to be processed
      while (curPos > 0) {
         try{
            wait();
         }
         catch (InterruptedException e){}
      }
      // store string
      contents = x;
      totalLength = x.length();
      curPos = totalLength;
      // notify any blocked receive calls
      notifyAll();
   }
}
```

Figure 1. Producer-consumer monitor

restricted to just this example, but will occur in most Java monitor implementations [8]. It is interesting to note that an early edition of a textbook on Java [13] incorrectly used a `notify` at the end of a producer-consumer implementation, but that this was replaced by `notifyAll` in a later edition.

4 Deterministic Testing of Java Monitors

The steps involved in testing a Java monitor are the same as in the original method, but the details in some of the steps have changed. Each of these is described in more detail below.

To document the method, we write a *Test Plan* for each monitor that we test. The sections in a Test Plan are *test conditions*, *special cases*, and *test sequence*.

4.1 Step 1: Identifying preconditions

In the original method, a set of preconditions are derived that will cause every branch of the monitor operations to be executed. However, since Java monitors typically contain while-conditions instead of if-conditions, it is more appropriate to consider loop coverage instead of branch coverage, which suggests that we should select sufficient test cases so that the loop is entered 0 times, 1 time, and multiple times.

In addition, instead of notifying (waking up) a single thread, Java monitors typically notify all suspended threads. This means that we should consider the number and types of processes suspended on the monitor queue for each call to `notifyAll`. Following common testing practice, we should include tests for when the queue is empty, contains one thread, or contains multiple threads. It is interesting to note that the monitor implementation presented in [2] does not handle the situation where multiple consumer threads are suspended on a call to `receive`, and the only way to detect this would be to include tests with multiple suspended consumer threads.

Finally, even though it is not part of the original method, we consider any special monitor state or parameter values that we want to test. For example, for the producer-consumer monitor, we should test that the implementation behaves correctly when we send an empty string. This is again because it is good testing practice.

Ideally, we would like to test all combinations of the cases above, but that would lead to a prohibitively large number of cases. Instead, we decide on which combinations of conditions to test and record this in the *test conditions* and *special cases* sections of the test plan.

These sections of the test plan for the producer-consumer monitor are shown in Figure 2. A unique identifier is included for each test condition and special case for later reference. In this case, we have 12 conditions/special cases that we want to test. In contrast, Brinch Hansen derived four conditions for the same monitor implemented in Concurrent Pascal.

test conditions

`receive()`		
	C_1	0 iterations of the loop
	C_2	1 iteration of the loop
	C_3	multiple iterations of the loop
`send()`		
	C_4	0 iterations of the loop
	C_5	1 iteration of the loop
	C_6	multiple iterations of the loop
processes suspended on the queue		
	C_7	no processes suspended
	C_8	one sender suspended
	C_9	one receiver suspended
	C_{10}	multiple senders suspended
	C_{11}	multiple receivers suspended

special cases

`send()`		
	C_{12}	empty string

Figure 2. Test conditions and special cases for producer-consumer monitor

4.2 Step 2: Constructing a sequence of calls

In the second step, the tester constructs a sequence of monitor calls that will exercise each of the test conditions and special cases identified in step 1. There are many sequences that will exercise all conditions, and the tester simply has to construct one of these (typically, through trial and error). The resulting test sequence is recorded in the *test sequence* section of the test plan.

Figure 3 shows the initial part of the test sequence for the producer-consumer monitor. Each call is identified by a unique time-stamp of the form T_i, and the calls are interspersed with annotations that record any conditions/special cases from the test plan that the following call satisfies, relevant aspects of the current state of the monitor variables, and a set of suspended monitor calls (each suspended call is identified by the time-stamp at which the call was made). The last two entries facilitate the checking of the test sequence against the test conditions and special cases.

In the example, the first call is `send("ab")`, which satisfies conditions C_4, 0 iterations of the loop for a call to `send`, and C_7, no processes suspended when `notifyAll` is called. It is important to note that when a process is wo-

test sequence

$[C_4, C_7;$ `contents` $=$ `""`, `curPos` $= 0; \langle\rangle]$

T_1: `send("ab")`

$[C_7;$ `contents` $=$ `"ab"`, `curPos` $= 2; \langle\rangle]$

T_2: `send("cd")`

$[C_1, C_8;$ `contents` $=$ `"ab"`, `curPos` $= 2; \langle T_2\rangle]$

T_3: `receive()`

$[C_1, C_8;$ `contents` $=$ `"ab"`, `curPos` $= 1; \langle T_2\rangle]$

T_4: `receive()`

$[C_5, C_7;$ `contents` $=$ `"ab"`, `curPos` $= 0; \langle T_2\rangle]$

T_5: T_2 woken up

$[-;$ `contents` $=$ `"cd"`, `curPos` $= 2; \langle\rangle]$

Figure 3. Part of test sequence for producer-consumer monitor

ken up, this is also recorded in the test sequence. For example, at time T_5, the thread that called `send("cd")` is woken up. With the four calls above, 5 out of 12 conditions/special cases are satisfied. The sequence that we used to satisfy all 12 conditions consists of 21 calls. This is in contrast with the 6 calls that Brinch Hansen needed.

4.3 Step 3: Implementing the sequence

For the testing to succeed, the execution of the sequence of calls must be kept in the same order as described in Figure 3. This means that we must implement a test driver that starts up a number of threads that call the monitor procedures in the prescribed order. However, the relative progress of these threads will normally be influenced by numerous unpredictable and irreproducible events, such as the timing of interrupts and the execution of other processes.

To guarantee the order of execution, an abstract clock class is introduced to provide synchronisation. This clock provides two operations: `await(`t`)` delays the calling thread until time t is reached, and `tick` advances the time by one unit, waking up any processes that are awaiting that time. Progression of time is controlled by a separate process, that makes the clock tick at regular intervals. The interval is chosen to be large enough to guarantee that any call or waking up of a test process is guaranteed to complete within one interval. This use and the implementation of the timer is identical to the original method.

To further support the method, we have adapted the Roast testing tool [6, 10, 9]. We have incorporated the clock class in Roast and used the Roast test templates to implement the individual calls to the monitor operations. The templates provide automatic checking of exceptions and a convenient way for checking the return values of monitor calls. In addition, Roast provides support for debugging when the testing reveals a failure.

```
class producer1 extends thread {
   private producerConsumer m;
   private monitorClock c;
   ...
   public void run() {
      c.await(1);
      #excMon m.send("ab"); #end
      c.await(2);
      #excMon m.send("cd"); #end
      c.await(5);
      ...
   }
}
class consumer1 extends thread {
   private producerConsumer m;
   private monitorClock c;
   ...
   public void run() {
      c.await(3);
      #valCheck m.receive() # 'a' #end
      c.await(4);
      #excMon m.receive() # 'b' #end
      ...
   }
}
```

Figure 4. Part of test driver for producer-consumer monitor

Continuing the example, Figure 4 shows part of two threads that are used in the test driver for the producer-consumer monitor (the full driver contains seven such threads). The calls in the methods correspond to the monitor calls in Figure 3. The translation from the test sequence to the test driver code is straightforward. Note that the call `await(5)` in `producer1` is there to allow the thread to complete the call to `send("cd")` at time T_2 that was suspended then.

Calls to `send` are placed inside a Roast exception-monitoring template (delimited by `#excMon` and `#end`) to ensure that no exceptions are thrown during the call. Similarly, the calls to `receive` are placed inside a Roast value-checking template (delimited by `#valCheck`, `#` and `#end`) to ensure that the call before the `#` returns the expected output after the `#`.

4.4 Step 4: Execution and comparison

With Roast, test case execution and output comparison has been automated and this step is trivial.

5 Testing of Readers-Writers

To experiment further with the method, we applied it to the readers and writers problem [13], which is the abstraction of the problem of separate processes accessing a shared resource (such as a file or database). A reader process is only allowed to examine the content of the resource, while a writer can examine and update the content. The problem is to ensure access to the resource so that multiple readers are allowed to examine the resource at the same time, while only one writer is allowed to update the resource at a time. Moreover, no readers should be allowed to examine the resource while a writer is accessing it.

We tested a typical solution to the readers and writers problem, which is a monitor with four monitor procedures:

- `startRead` is called by a reader that wants to start reading;
- `endRead` is called by a reader that is finished reading;
- `startWrite` is called by a writer that wants to start writing; and
- `endWrite` is called by a writer that is finished writing.

Since calls to `endRead` and `endWrite` should never suspend, only the calls to `startRead` and `startWrite` have calls to `wait` in them. Similarly, only calls to `endRead` and `endWrite` have calls to `notifyAll` in them.

Applying the enhanced method to the readers and writers problem proved relatively straightforward. However, there were a total of 29 test conditions, and a test sequence of 31 monitor calls was needed to satisfy these test conditions. The task of deriving the test sequence was non-trivial and this suggests that it might be better to come up with multiple shorter test sequences to cover the test conditions in the future.

To further evaluate our method, a series of eight erroneous versions of the readers and writers implementation were generated by seeding one error in each implementation. Unfortunately, when we ran the test driver, only three of the eight erroneous implementations were detected. Upon further investigation, we found that this was because we have no way to detect when a thread completes a call.

For example, even though in the implementation of the `producer1` thread in Figure 4, we have a call to `await(5)` to allow the `send("cd")` call to complete, this does not guarantee that this call did not complete at an earlier stage. In particular, the call to `send("cd")` might not have suspended at all, which would have been incorrect. In the producer-consumer case we would have detected this because it would have resulted in incorrect return values for `receive` at a later stage. However, a much better method would be to check the time at which a call is completed, especially in cases where a call is supposed to suspend.

To do this, we added output statements into the test driver to monitor when calls are completed. While this is not a particularly elegant solution, it did reveal the errors in the other five erroneous implementations. In the future, we will enhance the method and the clock class to allow us to detect when a monitor call is completed.

6 Conclusion and Future Work

The non-deterministic nature of concurrent programs means that conventional testing methods are inadequate for testing such programs. Deterministic execution is a strategy that is commonly used in the testing of concurrent programs, and it is used here in a method for testing Java monitors.

The test method is derived from an existing method [2] that tests Concurrent Pascal monitors. The original method provided insufficient test coverage when applied to Java monitors and was extended accordingly. The method consists of four steps: identifying preconditions, constructing a sequence of calls, implementing the sequence, and execution and comparison. The original method was extended in both the identification of suitable preconditions and in providing tool support.

The new method was applied to the original producer-consumer monitor and to the readers and writers problem. The application to the readers and writers brought out two minor problems. First, the number of test conditions was large and as a result it was hard to come up with one test sequence that satisfied all conditions. In the future, we plan to use a number of smaller sequences instead, although we would like to avoid having to write more than one test driver to accomplish this. Scaling up in general is a problem that needs to be looked at, but we note that the original method was applied to a real-time scheduler and a multicomputer network program [2].

Second, it proved necessary to be able to detect when monitor calls that are supposed to suspend, wake up. In the readers and writers example this was accomplished by including output statements in the test driver. In the future, we plan to augment the clock class and the test driver to detect this.

The other area for future work we are pursuing is the automated generation of a test driver from the test sequence(s). As explained, this is a fairly mundane step and we believe that much of this step can be automated by using a suitable notation for describing the test sequence(s).

Acknowledgements

Thanks to Adrian Birch, Dan Hoffman, Brad Long, and Tim Miller for their comments on earlier versions of this paper.

References

[1] M. Ben-Ari. *Principles of Concurrent and Distributed Programming*. Prentice Hall, 1990.

[2] P. Brinch Hansen. Reproducible testing of monitors. *Software-Practice and Experience*, 8:721–729, 1978.

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

[4] R. 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.

[5] J. Choi and H. Srinivasan. Deterministic replay of Java multithreaded applications. In *Proceedings of the Symposium on Parallel and Distributed Tools*, 1998.

[6] N. Daley, D. Hoffman, and P. Strooper. Unit operations for automated class testing. Technical Report 00-04, Software Verification Research Centre, The University of Queensland, Jan. 2000.

[7] P. Godefroid, L. Jagadeesan, R. Jagadeesan, and K. Laufer. Automated systematic testing for constraint-based interactive services. In *Proceedings of the ACM SIGSOFT 8th International Symposium on the Foundations of Software Engineering*, volume 25, 6 of *ACM Software Engineering Notes*, pages 40–49. ACM Press, 2000.

[8] S. Hartley. Alfonse, your Java is ready! In *Proceedings of SIGSCE'98*, pages 247–251. ACM Press, 1998.

[9] D. Hoffman and P. Strooper. Prose + test cases = specifications. In *Proceedings* 34^{th} *International Conference on Technology of Object-Oriented Languages and Systems*, pages 239–250. IEEE Computer Society, 2000.

[10] D. Hoffman and P. Strooper. Techniques and tools for Java API testing. In *Proceedings 2000 Australian Software Engineering Conference*, pages 235–245. IEEE Computer Society, 2000.

[11] T. Katayama, E. Itoh, and Z. Furukawa. Test-case generation for concurrent programs with the testing criteria using interaction sequences. In *Proceedings of the 2000 Asia-Pacific Software Engineering Conference*, pages 590–597. IEEE Computer Society, 2000.

[12] D. Long and L. Clarke. Data flow analysis of concurrent systems that use the rendezvous model of synchronisation. In *Proceedings of the Symposium on Software Testing, Analysis and Verification (TAV4)*, pages 21–35. ACM Press, 1991.

[13] J. Magee and J. Kramer. *Concurrency State Models and Java Programs*. John Wiley & Sons, 1999.

[14] G. Naumovich, G. Avrunin, and L. Clarke. Data flow analysis for checking properties of concurrent Java programs. In *Proceedings of the 1999 International Conference on Software Engineering*, pages 399–410. IEEE Computer Society, 1999.

[15] K.-C. Tai, R. Carver, and E. Obaid. Debugging concurrent Ada programs by deterministic execution. *IEEE Transactions on Software Engineering*, 17(1):45–62, 1991.

[16] R. Taylor. A general-purpose algorithm for analyzing concurrent programs. *Communications of the ACM*, 26(5):362–376, 1983.

[17] W. Yeh and M. Young. Redesigning tasking structures of Ada programs for analysis: a case study. *Software Testing, Verification and Reliability*, 4:223–253, 1994.