View metadata, citation and similar papers at core.ac.uk

brought to you by CORE

provided by University of Queensland eSpace

SOFTWARE VERIFICATION RESEARCH CENTRE

SCHOOL OF INFORMATION TECHNOLOGY

THE UNIVERSITY OF QUEENSLAND

Queensland 4072
Australia

TECHNICAL REPORT

No. 01-21

Tool Support for Testing Java Monitors

Brad Long Dan Hoffman
Paul Strooper*

June 2001

Phone: +61 7 3365 1003
Fax: +61 7 3365 1533
http://svrc.it.uq.edu.au

Note: Most SVRC technical reports are available via anonymous ftp, from `svrc.it.uq.edu.au` in the directory `/pub/techreports`. Abstracts and compressed postscript files are available via `http://svrc.it.uq.edu.au`

Tool Support for Testing Java Monitors

Brad Long* Dan Hoffman† Paul Strooper*

Abstract

The Java programming language supports monitors. Monitor implementations, like other concurrent programs, are hard to test due to the inherent non-determinism. This paper presents the ConAn (*Con*currency *An*alyser) tool for generating drivers for the testing of Java monitors. To obtain adequate controllability over the interactions between Java threads, the generated driver contains processes that are synchronized by a clock. The driver automatically executes the calls in the test sequence in the prescribed order and compares the outputs against the expected outputs specified in the test sequence. The method and tool are illustrated in detail on an asymmetric producer-consumer monitor, and their application to two other monitors is discussed.

1 Introduction

A Java monitor encapsulates data that can only be observed and modified by monitor access procedures [13]. Only one thread may be active inside a monitor at a time, giving each thread mutually exclusive access to the data encapsulated.

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. That is, if we run a concurrent program twice with the same test input, it is not guaranteed to return the same output both times. This is because some event orderings may vary between executions. This non-determinism causes two significant test automation problems: (1) it is hard to force the execution of a given program statement or branch and (2) it is difficult to automate the checking of test outputs.

In this paper, we extend a method for testing monitors proposed by Brinch Hansen [1]. 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 execute the monitor calls as defined in the previous step. 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.

*School of Computer Science and Electrical Engineering, Software Verification Research Centre, The University of Queensland, Brisbane, Qld 4072, Australia. email: { brad, pstroop }@csee.uq.edu.au

†Dept. of Computer Science, University of Victoria, PO Box 3055 STN CSC, Victoria, B.C. V8W 3P6, Canada. email: dhoffman@csr.uvic.ca

By using an external clock to synchronize the calls to the monitor, we can control the interaction of the test processes without changing the code under test and thus guarantee that we exercise the preconditions we want to.

The original method was devised for monitors implemented in Concurrent Pascal. In [7], we enhanced the method to test Java monitors. In particular, we extended the test selection criterion in the first step to include loop coverage, consideration for the number and type of processes suspended inside the monitor, and interesting state and parameter values. We also provided tool support by using the Roast tool for testing Java classes [5, 9, 8] to check exception behavior and output values of monitor calls.

In this paper, we provide further tool support through ConAn (*Con*currency *An*alyser), which automates the third step in the method. With ConAn, the tester specifies the sequence of calls and the threads that will be used to make those calls. From this information, ConAn generates a test driver that controls the synchronization of the threads through a clock and that compares the outputs against the expected outputs specified in the test sequence, including the time at which each monitor call should complete. The generated driver also detects when a process in a test sequence is suspended indefinitely.

We review related work in Section 2. In Section 3, we introduce an asymmetric producer-consumer monitor used to illustrate the method and ConAn. We describe the method in Section 4 and apply it to the asymmetric producer-consumer monitor. In Section 5, we describe the full functionality of ConAn. We then discuss its application to three example monitors in Section 6.

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 behavior of concurrent programs have been proposed [16, 12, 17, 14, 10]. The resulting graphs are then analyzed to generate suitable test cases, to generate suitable synchronization 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 [2, 15, 4] have proposed techniques for "replaying" concurrent computations. While helpful, such tools do nothing to achieve adequate test coverage. Carver and Tai [3] 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 [6]. The method uses an algorithm to systematically generate all possible behaviors of such a program, and these behaviors 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 [1], who 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.

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

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

Figure 1: Producer-consumer monitor

3 Producer-Consumer Monitor

The `ProducerConsumer` class shown in Figure 1 implements an asymmetric Producer-Consumer monitor, the Java equivalent of the Concurrent-Pascal program described in [1]. 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 monitor state is maintained through three variables: `contents` stores the string of characters, `curPos` represents the number of characters in `contents` that have yet to be received, and `totalLength` represents the length of `contents`.

The `synchronized` keyword in the declaration of the `send` and `receive` methods specifies that these methods must be executed under mutual exclusion, i.e., only one thread can be active inside one of these methods at any time. Thus, if thread T attempts to execute a synchronized method in a class while there is another thread active in the same class, T will be suspended.

The `wait` operation is used to block a consumer thread when there are no characters in the buffer, and a producer thread when the buffer is nonempty. It suspends the thread that executed the call and releases the synchronization lock on the monitor. Each `wait` call is placed inside a `try-catch` block to trap any thread interruption exceptions that may occur. The `notifyAll` operation wakes up all suspended threads, although only one thread at a time will be allowed to access the monitor.

`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 empty string
- C_5 0 iterations of the loop
- C_6 1 iteration of the loop
- C_7 multiple iterations of the loop

processes suspended on the queue

- C_8 no processes suspended
- C_9 one sender suspended
- C_{10} one receiver suspended
- C_{11} multiple senders suspended
- C_{12} multiple receivers suspended

Figure 2: *Test conditions* for producer-consumer monitor

4 Testing Java Monitors

The four steps involved in testing a Java monitor are described below.

4.1 Step 1: Identifying preconditions

In the original method [1], a set of preconditions is derived that will cause every branch of the monitor operations to be executed. Since Java monitors typically contain while-conditions, we aim to achieve loop-coverage of the code under test instead. Specifically, we select sufficient test cases so that each loop is executed 0, 1, and multiple times.

In addition, we consider the number and types of processes suspended on the monitor queue for each call to `notifyAll`. Following common testing practice, we include tests for queue size 0, 1, and greater than 1.

Finally, we consider any special monitor state or parameter values that we want to test. For the producer-consumer monitor, we test that, for example, the implementation behaves correctly when we send an empty string.

Ideally, we would 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 these as *test conditions*; see Figure 2. A unique identifier is included for each test condition for later reference. In this case, we have 12 conditions that we want to test.

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. The tester must construct one or more of these, typically through trial and error. One long test sequence is possible [7], but we have found that it is easier to manage multiple, short test sequences that each exercise one or more conditions.

Figure 3 shows a test sequence for the producer-consumer monitor. With each call is a unique time-stamp, the output produced by the call, the conditions that the call satisfies, and a set of suspended monitor calls, each identified by the time at which the call was made. The *calls suspended* column facilitates the checking of the test sequence against the test conditions.

time	call	output	conditions	calls suspended
T_1	`send("a")`	–	C_5, C_8	–
T_2	`send("b")`	–	C_9	T_2
T_3	`receive()`	`'a'`	C_6	–
T_4	`receive()`	`'b'`	–	–

Figure 3: A test sequence for producer-consumer monitor

In the example, the first call is `send("a")`, which satisfies both conditions C_5 (0 iterations of the loop) and C_8 (no processes suspended). We also consider when a process is woken up as part of the test sequence. At time T_3, after the `receive` call, the suspended monitor call, T_2, is woken up, hence satisfying condition C_6 (1 iteration of the loop for a call to `send`).

Note that we do not record all conditions satisfied by the calls. For example, we do not record that the `receive` call at time T_3 satisfies condition C_1, because this test sequence was designed to test the suspension and waking up of `send` calls.

4.3 Step 3: Implementing the sequence

During test execution, the call sequence must be as described in Figure 3. This means that we must implement a test driver that starts 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, the method uses an abstract clock to provide synchronization. This clock provides three operations: `await(`t`)` delays the calling thread until the clock reaches time t, `tick` advances the time by one unit, waking up any processes that are awaiting that time, and `time` returns the number of units of time passed since the clock started. The `time` operation has been added to the method to detect when threads wake up. Previously, for example, threads violating safety properties of the monitor and threads waking at incorrect times could go undetected. This could occur if the test case passed, based simply on the output of the monitor calls. The `time` call allows a tester to ensure each thread wakes up at a certain time or between a range of times.

ConAn automates this step in the method by allowing the tester to specify the sequence of monitor calls and by assigning each call to a thread. ConAn sets up the clock and timer, generates `await` calls to control the order of calls to the monitor, and manages the passing of time. Progression of time is controlled by a separate process that makes the clock tick at regular intervals. The time 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 time interval.

The script writer never needs to deal directly with the clock or timer. If a liveness error causes a thread to suspend indefinitely, ConAn detects this, terminates the thread at completion of the test sequence, reports an appropriate error message, and continues with the next test sequence. On completion of a test script, the number of test cases, the number of value errors, and the number of liveness errors are reported.

Continuing the example, Figure 4 shows a ConAn test sequence for the producer-consumer monitor. Two threads, `sender` and `receiver`, are used. The calls in the methods correspond to the monitor calls in Figure 3. The translation from the test sequence to the ConAn test script is straightforward. Each tick block, delimited by `begin_tick` and `end_tick`, calls a monitor command. The call to `send("a")` at time 1 (T_1) completes at time 1. Then, the call to `send("b")` at time 2 suspends and does not complete until time 3, when a call to `receive()` is made.

```
begin_case
    goal_conditions C5 C6 C8 C9
    begin_tick                  // T1
        begin_thread sender
            #excMonitor m.send("a"); #end
            #valueCheck time() # 1 #end
        end_thread
    end_tick
    begin_tick                  // T2
        begin_thread sender
            #excMonitor m.send("b"); #end
            #valueCheck time() # 3 #end
        end_thread
    end_tick
    begin_tick                  // T3
        begin_thread receiver
            #valueCheck m.receive() # 'a' #end
            #valueCheck time() # 3 #end
        end_thread
    end_tick
    begin_tick                  // T4
        begin_thread receiver
            #valueCheck m.receive() # 'b' #end
            #valueCheck time() # 4 #end
        end_thread
    end_tick
end_case
```

Figure 4: ConAn test sequence to *test a sender suspended*

To further support the method, we have integrated ConAn with the Roast testing tool [5, 9, 8] by allowing ConAn to include Roast test templates. We use 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.

Calls to `send` are placed inside a Roast exception-monitoring template, delimited by `#excMonitor` and `#end`, to ensure that no exceptions are thrown during the call. Similarly, calls to `receive` are placed inside a Roast value-checking template, delimited by `#valueCheck`, `#` and `#end`, to ensure that the call before the `#` returns the expected output after the `#`. After each monitor call, the clock `time` function is also called to check the time at which the thread completes the call.

4.4 Step 4: Execution and comparison

Test case execution and comparison is fully automated. The test script is parsed by Roast and ConAn, producing a driver as a Java source code file. Then the driver is compiled and executed.

5 ConAn Syntax and Semantics

Figure 5 shows the general structure of a ConAn test script. Each section of the test script is described below.

- `driver` ⟨*driver-name*⟩: *driver-name* will be used as the name of the generated driver program, i.e., the name of the Java class required to run the test sequences.

```
driver ⟨driver-name⟩
monitor ⟨class-name⟩ ⟨monitor-id⟩
begin_conditions
     ⟨condition-id⟩ ⟨condition-description⟩
     ...
end_conditions
begin_case
     goal_conditions ⟨condition-id⟩ ...
     begin_setup
          ⟨Java code⟩ // test sequence setup code (if any)
     end_setup
     begin_tick
          begin_thread ⟨thread-id⟩
               ⟨Java code⟩ // code for this thread
          end_thread
          begin_thread ⟨thread-id⟩
               ⟨Java code⟩ // code for this thread
          end_thread
          ...
     end_tick
     begin_tick
          ...
     end_tick
     ...
     begin_teardown
          ⟨Java code⟩ // test sequence teardown code (if any)
     end_teardown
end_case
begin_case
     ...
end_case
...
exit
```

Figure 5: Structure of a ConAn test script

- `monitor` ⟨*class-name*⟩ ⟨*monitor-id*⟩: *class-name* is the name of the monitor under test (MUT). The *monitor-id* is any valid Java variable name and is used to create an instance of the MUT. This identifier can then be used in the test sequences to reference the MUT instance. A new MUT instance is created for each test sequence.

- ⟨*condition-id*⟩ ⟨*condition-description*⟩: Conditions are listed in the conditions block, delimited by `begin_condition` and `end_condition`. Each condition is identified by *condition-id*; the *condition-description* is plain text. All conditions are documented in this section and may be referenced in the `goal_conditions` section of the test sequences. Conditions are an aid to understanding and evaluating a test script and provide traceability between the test sequences and the listed conditions. Conditions that do not appear in the `goal_conditions` list of any test sequence are reported by ConAn as potential problems. However, the use of conditions in a test script is optional.

- Test sequence: A test script consists of one or more test sequences, delimited by `begin_case` and `end_case`. Each test sequence consists of a number of tick blocks representing units of time (*ticks*) of length *tickTime* seconds. A test sequence completes after $n + 1$ ticks have passed, where n is the number of tick blocks in the sequence.

- `goal_conditions` ⟨*condition-id*⟩: For each test sequence, a number of goal conditions may be listed. Conditions may be listed in more than one test sequence, so condition references are not required to be unique across test sequences.

- Setup and Teardown blocks: Setup code, delimited by `begin_setup` and `end_setup`, and teardown code, delimited by `begin_teardown` and `end_teardown`, can be inserted at the beginning and end of a test sequence. Each thread within a test sequence is an instance of the same Java class. This means that an instance of code and variables created in the setup block is defined for each thread in the test sequence. However, only one instance is created for variables defined as `static`, which is shared by all threads in the test sequence.

- Tick block: Each tick block, delimited by `begin_tick` and `end_tick`, represents a unit of time (or *tick*). Each tick has a duration of *tickTime* seconds. It is assumed that any statement executing within a tick block will complete before *tickTime* seconds has passed. Hence, all statements, for all threads within a tick block, execute within the same unit of time.

- Thread block: Each thread block begins with the `begin_thread` statement that identifies the thread that will execute the enclosed Java code by the thread identifier *thread-id*. For each unique thread identifier across all tick blocks in a test sequence, a separate thread is created. A thread with identifier *id* executes code for each tick block T that *id* appears in. The code that is executed first suspends the thread until time T is reached, and then executes the Java code associated with *id* and time T. Each thread identifier may appear only once in each tick block.

  Any Java code or Roast test cases can be entered between the `begin_thread` and `end_thread` statements. In addition, after all ticks for a test sequence have completed, the driver checks to make sure there are no suspended threads for that test sequence.

6 Examples

6.1 Complete producer-consumer test script

Figure 6 presents some basic statistics about the full producer-consumer test script and the two other test scripts discussed in this section. Five test sequences were developed for testing

metric	producer-consumer	ConAn clock	readers-writers
Test Conditions	12	6	15
Test Sequences	5	3	8
Test Cases	36	16	84
Test Script LOC	150	40	305
Java Driver LOC	920	480	1855

Figure 6: Summary of test driver generation for example monitors

the twelve conditions of the producer-consumer monitor. The ConAn test script comprised 150 lines, containing 36 test cases. Of the 36 test cases, 18 were monitor calls and 18 were wakeup checks. The generated Java program comprised 920 lines.

This compares with the 21 monitor calls previously used to satisfy the same 12 conditions [7]. In that case, however, the driver required 200 lines of Roast script and produced 500 lines of Java code. Although ConAn generated more lines of Java code, the ConAn script is much simpler than the 200 line Roast script. Moreover, it contains all test conditions (these were recorded in a separate test plan in [7]) and additional tests to check the time at which monitor calls complete.

6.2 Testing the Clock monitor

The clock that is used by ConAn is also a monitor. We decided to test the clock monitor using ConAn. The test script for the clock monitor was simple to create from the conditions listed in Figure 7.

`await()`
- C_1 0 iterations of the loop
- C_2 1 iteration of the loop
- C_3 multiple iterations of the loop

processes suspended on the queue
- C_4 no processes suspended
- C_5 one awaiter suspended
- C_6 multiple awaiters suspended

Figure 7: *Test conditions* for the clock

One of the three test sequences is shown in Figure 8. Note that the calls to `m.time()` in the second tick block refer to the time function of the monitor under test (MUT). This can be identified by the use of the monitor variable `m`. The calls to `time()` without a qualifying variable refer to ConAn's clock. When the second tick block completes, ConAn's time has advanced to 2. However, the MUT's time is only incremented when we call `m.tick()`. Hence, the check for time 0 before the call to `m.tick()` and time 1 after the call.

6.3 Testing a Readers-Writers monitor

To experiment further with ConAn, we applied it to the readers and writers problem [13], which is an 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

```
begin_case
    goal_conditions C2 C5
    begin_tick                  // T1
        begin_thread awaiter
            #excMonitor m.await(1); #end
            #valueCheck time() # 2 #end
        end_thread
    end_tick
    begin_tick                  // T2
        begin_thread ticker
            #valueCheck m.time() # 0 #end
            #excMonitor m.tick(); #end
            #valueCheck m.time() # 1 #end
            #valueCheck time() # 2 #end
        end_thread
    end_tick
end_case
```

Figure 8: A test sequence for the clock monitor

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 ConAn to the readers and writers problem proved relatively straightforward. We created 8 sequences, with a total of 42 monitor calls, to test 15 conditions. The construction of each test sequence was straightforward, compared with the non-trivial exercise of creating one long sequence consisting of 31 monitor calls [7]. Using multiple shorter test sequences has greatly simplified the selection of test cases to cover the test conditions.

One minor problem that we encountered was with the non-determinism of waking up threads in Java. When multiple threads are suspended and are waiting on the same monitor entry condition, Java does not specify the order in which these suspended threads are woken up. In fact, we found that this order is not the same for different platforms on which we ran the tests. To make the tests platform-independent, we changed the test cases to check for one of two possible wake-up times in one thread and the other wake-up time for the second thread.

To further evaluate ConAn, we used the same seven faulty mutant versions of the readers-writers monitor implementation that were used in [7]. In that paper, only three faulty mutants were detected. By implementing thread suspension handling and the new `time` function, we were successful in detecting all seven faulty mutants.

7 Conclusion

The non-deterministic nature of concurrent programs means that conventional testing methods are inadequate. 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 [1] that tests Concurrent Pascal monitors. The method consists of four steps: identifying preconditions, constructing a sequence of calls, implementing the sequence, and execution and comparison. In earlier work [7], the method was extended in the area of identifying preconditions that are more suitable for Java monitors and in providing basic tool support.

In this paper, we provide further tool support through ConAn, which automates the third step in the method. In addition, we implemented a time function in the clock, to check when calls wake up, and provide the ability to detect liveness errors. We also improved the method for monitor testing by using multiple shorter test sequences, rather than one long sequence. We discussed the application of ConAn to three monitors: Producer-Consumer, the ConAn clock, and Readers-Writers. It has also been used for testing a monitor in a commercial distributed system [11].

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. Replay and testing for concurrent programs. *IEEE Software*, 8(2):66–74, 1991.

[3] 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.

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

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

[6] 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.

[7] C. Harvey and P. Strooper. Testing Java monitors through deterministic execution. In *Proceedings of the Australian Software Engineering Conference*, 2001.

[8] D.M. Hoffman and P.A. 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.

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

[10] 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.

[11] B. Long and P. Strooper. A case study in testing distributed systems. In *Proceedings of the 3rd International Symposium on Distributed Objects and Applications*, 2001.

[12] D. Long and L.A. 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.H. Carver, and E. Obaid. Debugging concurrent Ada programs by deterministic execution. *IEEE Transactions on Software Engineering*, 17(1):45–62, 1991.

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

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