Tool Support for Testing Concurrent Java Components

Brad Long, *Member*, *IEEE Computer Society*,
Daniel Hoffman, *Member*, *IEEE*, and Paul Strooper, *Member*, *IEEE*

Abstract—Concurrent programs are hard to test due to the inherent nondeterminism. This paper presents a method and tool support for testing concurrent Java components. Tool support is offered through ConAn (*Con*currency *An*alyser), a tool for generating drivers for unit testing Java classes that are used in a multithreaded context. To obtain adequate controllability over the interactions between Java threads, the generated driver contains threads 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. Their application to testing over 20 concurrent components, a number of which are sourced from industry and were found to contain faults, is presented and discussed.

Index Terms—Component testing, concurrency, unit testing, tool support.

1 Introduction

A concurrent program specifies two or more processes (or threads) that cooperate in performing a task [1]. Each process is a sequential program that executes a sequence of statements. The processes cooperate by communicating using shared variables or message passing.

The testing of concurrent programs is difficult due to the inherent nondeterminism 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 nondeterminism 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.

Unit testing is becoming an increasingly popular approach for assisting the development of high quality software. In particular, extreme programming and other agile development methods advocate a strict approach to unit testing as part of the software development process [5], [34]. While unit testing tools exist for sequential programs, there are currently no tools for the systematic unit testing of concurrent software components.

In this paper, we extend a method for testing monitors proposed by Brinch Hansen [7] and introduce tool support for unit testing concurrent Java components. Brinch Hansen's original method only applied to monitors. A monitor encapsulates data and the procedures that operate upon them [6]. A monitor not only protects internal data from unrestricted access, but also synchronizes calls to the interface procedures. This ensures that only one thread may be active inside a monitor at a time, giving each thread mutually exclusive access to the encapsulated data.

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.

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 [25], 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 threads suspended inside the monitor, and interesting state and parameter values. We also provided tool support by using the Roast tool for testing Java classes [16], [32] to check exception behavior and output values of monitor calls.

In this paper, we provide tool support through ConAn (*Con*currency *An*alyser), which automates the third step in the method. We demonstrate the use of ConAn for detecting faults in a range of concurrent components, in addition to

- *B. Long and P. Strooper are with the School of Information Technology and Electrical Engineering, Software Verification Research Centre, The University of Queensland, Brisbane, Qld 4072, Australia. E-mail: {brad, pstroop}@itee.uq.edu.au.*
- *D. Hoffman is with the Department of Computer Science, University of Victoria, PO Box 3055 STN CSC, Victoria, B.C. V8W 3P6, Canada. E-mail: dhoffman@csr.uvic.ca.*

Manuscript received 3 Apr. 2002; revised 14 Aug. 2002; accepted 7 Mar. 2003.
Recommended for acceptance by A. Bertolino.
For information on obtaining reprints of this article, please send e-mail to: tse@computer.org, and reference IEEECS Log Number 116223.

monitors, and we reduce the original method to three steps. 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 call should complete. The generated driver also detects when a thread in a test sequence is suspended indefinitely.

Although our method and ConAn were originally devised for testing Java monitors, they can be used to test other Java components as well. Following Szyperski [53], we take a software component to be a unit of composition with contractually specified interfaces and explicit context dependencies. Such a component is likely to come to life through objects and, therefore, would normally consist of one or more classes. Each class is confined to a single component. To be able to test a component with ConAn, we need to be able to access the component through a call-based interface. Moreover, ConAn is specifically designed to test components that are accessible by multiple threads and where thread synchronization and suspension play a role.

The primary aim of this work has been to produce a practical method and tool support for the effective testing of concurrent Java components. By extending the techniques and tools for testing sequential Java components, the resulting framework can be used by software developers and testers in industry without the need for special training or advanced theoretical concepts. We demonstrate the practicality of the approach by discussing its application to the testing of over 20 concurrent components, sourced from textbooks, student assignments, and industry.

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. We then discuss the results of applying the method and tool to twenty concurrent components sourced from academia and industry, some of which were found to contain faults, in Section 5. The syntax and semantics of ConAn are detailed in the Appendix.

2 Related Work

2.1 Static Analysis

Several strategies for the testing of concurrent programs have been proposed in the literature. Static analysis typically involves the generation and analysis of (partial) models of the states and transitions of a program [35], [36], [42], [47], [55], [56]. The models are 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 models are large and complex. In many cases, this problem is compounded by a lack of tool support.

For example, Taylor [55], [56] examines concurrency states. A concurrency state summarizes the control state of each of the concurrent tasks at some point in an execution, including synchronization information, while omitting other information such as data values. Each concurrency state shows the next synchronization-related activity to occur in each of the system's tasks. Each state has zero or more successor states. The list of states and their successor states represents the possible interactions that can occur across all executions of a system. A concurrency history is defined as a sequence of concurrency states. A concurrency state graph structure can be created from all possible concurrency states. A concurrency history is then a sequence of states lying on some path within the concurrency state structure. The graph of all possible interactions can quickly lead to state space explosion as many concurrency histories exist. In addition, dynamic creation of tasks makes it difficult to predict the state space. Due to these issues, some static analysis techniques ignore dynamic task creation entirely.

Other static analysis methods [17], [35], [36], [42] require the separation of a program into task (or event) interaction graphs (TIGs). A task interaction concurrency graph (TICG) is constructed from TIGs and represents the possible interactions among the TIGs. However, the TICG shows all concurrent states, including infeasible ones. Algorithms to determine task interaction graphs from source code, to assist with generation of test cases from graph representations, cannot distinguish between feasible and infeasible interactions. Dealing with such infeasible paths and with complex data structures are inherent limitations of data flow analysis. Although the purpose of some of these methods is to support the generation and execution of test cases, we have classified these papers under static analysis because they mostly describe the analysis aspects of the method.

Static analysis has been combined with other techniques, such as symbolic execution [58], in an attempt to overcome some of these shortcomings.

2.2 Model Checking

A model is a simplified representation of the real world. It includes only those aspects of the real-world system relevant to the problem at hand. Models of software are often based on finite state machines or call graphs with well-defined mathematical properties [13], [45]. This approach facilitates formal analysis and mechanical checking, thus avoiding the tedium (and introduction of errors) inherent in manual formal methods. Traditionally, a model of the program is created manually, in the form of a mathematical specification. For the last few years, models have been successfully generated automatically from the program source or object code. The remainder of this section focuses on the model checking of Java programs.

Proposals have been made to reduce the size of finite state models in Java programs to assist with development of tools that extract models from program source [15], [38]. Some model checking tools are now supporting dynamic data structures for modeling object creation [50].

One of the challenges of model checking is constructing an accurate and sufficient model from which to analyze desired properties [14]. Some model checkers include runtime analysis to provide an execution history of a program and then perform static analysis on the execution history. By considering an individual execution trace, the state space is reduced and errors in the program that may not otherwise be found by dynamic analysis can be checked by analyzing the static trace information. The program trace may have enough information such that potential errors are detected that did not occur during the run that collected the information. Obviously, paths that are not executed in a particular run of the program will not appear in the execution history, so important information may be missing making it unlikely that all potential errors will be detected. This approach is taken by the Eraser algorithm [51] to detect potential data races, and the LockTree [57] and GoodLock [28] algorithms to detect potential deadlocks.

The Bandera toolset [14], [26] takes as input Java source code and a formalized software requirement and generates a program model and specification in the input language of one of several existing model-checking tools. At this point in time, effective use of Bandera requires knowledge of temporal logic, explicit state model-checking, and abstract interpretation.

JPF [27], [57] combines model-checking techniques with techniques for dealing with large or infinite spaces. JPF uses state compression to handle complex states and partial order reduction, abstraction, and runtime analysis techniques to reduce state space. JPF runtime analysis uses the Eraser [51] and GoodLock algorithms as guides to the model checker [28]. JPF has been used with Bandera to take advantage of program slicing techniques.

FSP [43] is an algebraic notation used to describe process models. FSP comes from a family of notations including CSP [31] and CCS [46], and is designed to be easily machine-readable. LTS (Labeled Transition System) is a state-machine representation of an FSP process and can be analyzed using the LTSA model-checking tool.

Jlint [2] is a static program checker that is fast in execution and requires no special tool knowledge to operate. However, during analysis, many warnings are reported that are not necessarily errors. This makes review of warnings important but tedious and, hence, fault diagnosis can be error-prone. Possible deadlock review is difficult unless one has intimate knowledge of the software being checked.

2.3 Dynamic Analysis

Deterministic testing of concurrent programs requires forced execution of the program according to an input test sequence. This can be done by modifying the implementation of the synchronization constructs, controlling the runtime scheduler during execution, applying a source transformation strategy, or creating a test harness to control synchronization events without any modification to the software under test. Implementation-based approaches, such as those that modify some or all components of the compiler or runtime system [12], [37], do not require modification to the application code but can introduce portability issues, or even worse, runtime faults.

Language-based approaches are more portable, and the development of feasibility and sequence collection information is independent of implementation of the language. Brinch Hansen [7] 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. This method has the advantage that the monitor code under test need not be modified to provide deterministic execution. Harvey and Strooper [25] extended the method for Java to include loop coverage and the number and type of threads suspended.

Carver and Tai [9] apply a language-based approach to definition, collection, and feasibility problems of concurrent Ada programs. They collect concurrency histories of running programs and determine whether test sequences are feasible for a given input. They do not consider the selection of sequences that are effective for error detection. Feasibility of sequences is determined by transforming the original Ada program into another program that detects feasibility issues given certain inputs. Collection of sequences is similarly achieved by transforming the original program into one that records concurrency history information. In further work [11], they use a constraint-based approach to testing concurrent programs, which involves deriving a set of validity constraints from a specification of the program, performing nondeterministic 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. 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. Similar approaches utilizing program transformation techniques include the design of a toolset for dynamic analysis of Java programs [4] and frameworks and tools for testing concurrency in multithreaded and distributed systems [3], [8], [18].

Monitoring is the process of gathering information about a program's execution. Debugging is the process of locating, analyzing, and correcting suspected faults. The classical approach to debugging sequential programs involves repeatedly stopping the program during execution, examining the state, and then either continuing or re-executing in order to stop at an earlier point in the execution. Approaches to monitoring and debugging have been extended to include concurrent and distributed programs [29], [30], [33], [44], [49], [54]. Although these are important topics, we do not deal with them in this paper. A number of authors [10], [12], [20], [37] have proposed techniques for "replaying" concurrent computations. While helpful, such tools do nothing to achieve adequate test coverage. There are several tools and techniques for testing sequential Java classes and EJB components [16], [19], [21], [32], [48].

In contrast to the above approaches, we use a language-based dynamic approach that requires no code transformation or instrumentation and does not require a modified compiler or runtime system. ConAn cannot analyze execution traces for potential data races and deadlocks that did not occur in a test run, but relies on an independent test case selection strategy to build test sequences that attempt to detect those faults. Many model checking and static analysis tools tend to report false alarms or potential errors. ConAn reports only real errors that occur during execution.

```java
class ProducerConsumer {
    String contents;
    int charsRemaining = 0;
    int totalLength;

    // receive a single character
    public synchronized char receive() {
        char y;

        // wait if no character is available
        while (charsRemaining == 0) {
            try { wait(); }
            catch (InterruptedException e) {}
        }

        // retrieve character
        y = contents.charAt(totalLength - charsRemaining);
        charsRemaining = charsRemaining - 1;

        notifyAll(); // notify any blocked send/receive calls
        return y;
    }

    // send a string of characters
    public synchronized void send(String x) {

        // wait if there are more characters
        while (charsRemaining > 0) {
            try { wait(); }
            catch (InterruptedException e) {}
        }

        // store string
        contents = x;
        totalLength = x.length();
        charsRemaining = totalLength;

        notifyAll(); // notify any blocked receive calls
    }
}
```

Fig. 1. Producer-consumer monitor.

3 Producer-Consumer Monitor

The `ProducerConsumer` class shown in Fig. 1 implements an asymmetric Producer-Consumer monitor, the Java equivalent of the Concurrent-Pascal program described in [7].

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, `charsRemaining` 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 an object while T' is active in the same object, 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
threads suspended on the queue	
C_8	no threads suspended
C_9	one sender suspended
C_{10}	one receiver suspended
C_{11}	multiple senders suspended
C_{12}	multiple receivers suspended

Fig. 2. Test conditions for producer-consumer monitor.

4 Testing Concurrent Components

The three steps involved in testing a concurrent Java component are described below.

4.1 Step 1: Identifying Preconditions

The first step in the method is to identify preconditions that will cause the component under test to be exercised as desired. The intended operation of the component must be considered to determine which preconditions are sensible for the component. The method is independent of how these *test conditions* are chosen. Most components we have tested have been Java monitors, where we have used a combination of black-box and white-box techniques, taking the following three considerations into account.

1. Since Java monitors typically contain while-conditions surrounding `wait` calls, we aim to achieve loop-coverage of the code under test. Specifically, we select sufficient test cases so that each loop is executed 0, 1, and multiple times.
2. In addition, we consider the number and types of threads suspended on the monitor queue for each call to `notifyAll`. This is achieved by forcing threads to execute the `wait` call within the loop, thus suspending the appropriate threads. Then, `notifyAll` will notify the suspended threads. Following common testing practice, we include tests for 0, 1, and greater than 1 threads suspended on the wait queue.
3. Finally, we consider any special monitor state or parameter values that we want to test. For example, for the producer-consumer monitor, we test that the implementation behaves correctly when we send an empty string. Other traditional tests include testing for a string length of one and greater than one. However, we have chosen to focus on testing concurrency properties for this example.

Fig. 2 shows the test conditions that we have chosen for the producer-consumer monitor. A unique identifier is included for each test condition for later reference.

TABLE 1
A Sequence of Monitor Calls for Producer-Consumer

time	thread	call	output	conditions	call completion
1	T_1	`send("a")`	–	C_5, C_8	[1,2)
2	T_2	`send("b")`	–	C_9, C_6	[3,4)
3	T_3	`receive()`	'a'	–	[3,4)
4	T_4	`receive()`	'b'		[4,5)

4.2 Step 2: Constructing a Sequence of Calls

In the second step, the tester constructs a set of test sequences of monitor calls. The test sequences are designed to satisfy the test conditions identified in Step 1. One long test sequence is possible [25], but our experience with ConAn has shown that it is easier to manage multiple, short test sequences that each exercise one or more conditions.

As shown below, sequences can be written directly in a ConAn test script. To illustrate the concept, Table 1 shows the monitor calls in a test sequence for a producer-consumer. Each call in the test sequence is associated with a unique time stamp, shown in the first column. To be able to distinguish between the calls, we introduce a thread identifier in the second column. The monitor call is shown in the third column, with the expected output for that call in the fourth column. The fifth column shows the test conditions that the call satisfies; the sixth column shows call completion time.

In the example, the first call is send("a"), which satisfies both conditions C_5 (0 iterations of the loop) and C_8 (no threads suspended). The call is made at time 1. Since the call should complete without being suspended, it will complete during time 1 (and before time 2), designated by [1,2) in the call completion column. The terminology $[x, y)$ represents a call completing at or after time x, but before time y. At time 2, a second call to send is made. Since there is already a string stored in the producer-consumer monitor, thread T_2 should suspend. It will not complete until time 3 (but before time 4) as indicated by [3,4) in the final column of the table. Because there is now one sender suspended, this call satisfies condition C_9. At time 3, the receive call should complete and return the character 'a'. The last statement executed by thread T_3 before returning from the receive call is notifyAll. This wakes up thread T_2, which made the send call at time 2. Note that this thread will not get access to the monitor until the receive call completes and thread T_3 has released the lock on the monitor. After this, the call to send by T_2 completes, which means that the calls by both T_2 and T_3 should complete in the time range [3,4). Note also that as a result the send call by T_2 satisfies condition C_6 because by the time it completes it will have executed one iteration of the loop. Finally, the second call to receive should return the character 'b'.

To simplify test sequence construction and verification and to clarify the purpose of each test sequence, we do not record all conditions satisfied by the calls. For example, we do not record that the receive call at time 3 satisfies condition C_1 because this test sequence was designed to test the suspension and waking up of send calls. Another test sequence will cover that condition.

Test execution requires a test driver that starts a number of threads that call the monitor in the order prescribed in Table 1. 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 threads.

To guarantee the order of execution, the method uses a clock class that defines an abstract notion of time to provide the necessary synchronization. The time starts at zero and can only be incremented by a call to the method tick. The clock can also be used to suspend threads until its time reaches a certain value and to check the current value of time. In particular, the clock class provides three methods:

- await(t) delays the calling thread until the clock reaches time t,
- tick() advances the time by one unit, waking up any threads that are awaiting that time, and
- time() returns the current time t.

We have added the time method to allow us to determine the time at which calls complete. Otherwise, we cannot detect threads that wake up at incorrect times. The time method allows the tester to ensure each thread wakes up at a certain time or between a range of times.

Rather than manually implementing a complex test driver, we enter the test sequences directly into a ConAn script. ConAn generates the Java test driver code which sets up the clock, instantiates the threads for each test sequence, generates await calls to control the order of calls by each thread to the monitor, and manages the passing of time. Progression of time is controlled by a separate thread that invokes tick at regular intervals. The time interval is chosen to be large enough so that any call or waking up of a test thread should 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 terminates the thread at completion of the test sequence, reports the error, and continues with the next test sequence. On completion of a test script, the number of test cases and the number of errors are reported.

To further support the method, we have integrated ConAn with the Roast unit testing tool [16], [32] by allowing ConAn scripts to contain Roast test templates. We use the templates to provide automatic checking of exceptions and return values.

Continuing the example, Fig. 3 shows the ConAn test sequence corresponding to the monitor calls in Table 1. The test script normally consists of more than one sequence. The monitor variable m is declared previously in the script by

```
#begin
    #test C5 C6 C8 C9    // conditions exercised
    #tick                // time 1
        #thread
            #excMonitor m.send("a"); #end
            #valueCheck time() # 1 #end
        #end
    #end
    #tick                // time 2
        #thread
            #excMonitor m.send("b"); #end
            #valueCheck time() # 3 #end
        #end
    #end
    #tick                // time 3
        #thread
            #valueCheck m.receive() # 'a' #end
            #valueCheck time() # 3 #end
        #end
    #end
    #tick                // time 4
        #thread
            #valueCheck m.receive() # 'b' #end
            #valueCheck time() # 4 #end
        #end
    #end
#end
```

Fig. 3. A ConAn test sequence for producer-consumer.

the *component under test* command as an instance of the producer-consumer monitor, and it is freshly instantiated by the generated test driver for each test sequence (see the `#cut` command in the Appendix).

The translation from the table to the ConAn test script is straightforward. The conditions being exercised by the sequence are listed after `#test`. Conditions are an aid to understanding and evaluating a test script and provide traceability between the test sequences and the listed conditions (see the `#conditions` command in the Appendix). Conditions that do not appear in the `#test` list of any test sequence are reported by ConAn as potential problems. Each tick block, delimited by `#tick` and `#end`, contains a monitor call. The calls to `send` and `receive` are placed inside an exception-monitoring template, delimited by `#excMonitor` and `#end`, to ensure that no exceptions are thrown during these calls. The call to `send("a")` at time 1 should complete at time 1, which is checked by checking the return value of `time` after the call to `send` in a value-checking template, delimited by `#valueCheck`, `#`, and `#end`. The call to `send("b")` at time 2 suspends and does not complete until time 3, after the call to `receive()` has been made. These templates are expanded into `try-catch` blocks by the tool, thus removing the need for error handling code to be written explicitly by the tester.

4.3 Step 3: Execution and Comparison

Test case execution and comparison is fully automated. The test script is parsed by ConAn, producing a driver as a Java source code file. When the generated driver is compiled and executed, it runs the test sequences defined in the ConAn script and reports any detected errors as well as summary statistics.

Typically, both the test script generation time and the test case execution time are small. For example, on a Pentium II 350MHz desktop computer, it takes less than three seconds to generate the Java driver from a test script of 400 lines organized into ten test sequences. It then takes less than 10 seconds to execute the generated test driver, although this execution time clearly depends on the component under test.

The generated Java code relies heavily on the features of the Java `Thread` class, but otherwise does not use any special features of the JVM or Java APIs. As a result, ConAn should be easy to port to other versions of Java.

5 Experience

5.1 Overview

Table 2 shows an overview of components tested by ConAn. The components come from a variety of sources including industry, text books, and student assignments. The table lists the following information:

- Component: the name of the concurrent component under test.
- Cond: the number of test conditions.
- Seq: the number of test sequences required to test those conditions.
- Test Cases: the number of individual test cases.
- LOC CUT: the number of lines of code in the component under test.
- LOC ConAn Script: the number of lines of code in the ConAn script.
- LOC Gen. Driver: the number of lines of Java code in the generated test driver.
- Prepare: the time taken to prepare the test script (in minutes).
- Faults: the number of faults found.

By using the ConAn scripting language, the tester is spared from writing many lines of driver code. This is demonstrated by comparing the number of lines of test script (LOC ConAn script) to the number of lines of driver code (LOC Gen. Driver) in Table 2. The scripts are easy to prepare, taking up to 60 minutes to write. In many cases, significant portions of the test scripts can be reused from previous components tested, which further reduces preparation time. Even though the ConAn scripts are reasonably long in comparison to the lines of code of the component under test (LOC CUT), they do not grow significantly with the more complex components used in industry (components 15–20). This is because we are focusing on the concurrency properties of the components. The scripts would be longer if we were attempting to test all functional properties of the code under test. It is also worth mentioning that the ConAn syntax is quite verbose. We chose this syntax for readability. The number of lines of script would be reduced significantly if we had chosen a terse syntax.

The ProducerConsumer monitor is used in this paper as the example to demonstrate ConAn. Component 2 is the ConAn clock, which is used to control the execution order of ConAn test sequences. ReaderWriter, which controls access to resources, is a common monitor example. This

TABLE 2
Components Tested by ConAn

#	Component	Cond.	Seq.	Test Cases	LOC CUT	LOC ConAn Script	LOC Gen. Driver	Prep. (mins)	Faults
1	ProducerConsumer	12	5	36	38	150	920	40	
2	ConAnClock	6	3	18	36	45	511	15	
3	ReaderWriter	15	8	84	45	334	1959	60	
4	ReadWriteFair	15	9	88	37	236	2035	30	
5	ReadWritePriority	16	10	100	35	411	2250	20	1
6	ReadWriteSafe	15	9	88	33	236	2035	15	
7	FairBridge	19	8	80	40	215	1843	40	
8	SafeBridge	17	6	48	30	143	1179	10	
9	BufferImpl	13	6	52	39	132	1307	15	1
10	Semaphore	6	3	16	32	56	480	15	
11	CarParkControl	13	5	34	33	108	895	20	
12	SimpleAllocator	6	3	14	24	52	446	15	
13	FairAllocator	8	5	34	32	98	905	20	
14	BoundedOvertakingAllocator	9	6	54	37	220	1317	60	
15	BlastServer	6	3	16	282	45	480	15	
16	BlockingTerminalPool	15	10	106	929	420	2263	50	2
17	JDBCConnectionPool	6	3	18	282	50	480	25	1
18	ConnectionBroker	8	5	36	821	166	952	50	
19	ObjectPool	8	5	36	375	154	934	40	
20	RecordInputStream	12	8	74	185	260	1819	60	3

monitor is further described in this section; we present the detection capabilities of ConAn against a number of mutant implementations of this monitor. Components 4-14 are all sourced from [43]. Components 15-20 are sourced from industry. Specifically, Component 15 is used in a commercial middleware application [41]. It is a variation of the producer-consumer monitor. Components 16-20 are used in a commercial e-commerce product developed by a company which builds secure internet payment systems for integration with electronic commerce applications. The products are primarily server-based and use multithreaded programming techniques.

5.2 Producer-Consumer

Table 2 presents some basic statistics about the full producer-consumer test script. Five test sequences were developed for testing the 12 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 [25]. 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 [25]) and additional tests to check the time at which monitor calls complete.

5.3 Testing the ConAn Clock

The clock that is used by ConAn is a monitor. The test script for the clock monitor was simple to create from the conditions listed in Fig. 4. One of the three test sequences is shown in Fig. 5. Note that the calls to `m.time()` in the second tick block refer to the time function of the component under test (CUT). This can be identified by the use of the variable `m`, declared previously in the script as an instance of the CUT. The calls to `time()` without a qualifying variable refer to ConAn's clock. When the

```
await()
        C1   0 iterations of the loop
        C2   1 iteration of the loop
        C3   multiple iterations of the loop
threads suspended on the queue
        C4   no threads suspended
        C5   one awaiter suspended
        C6   multiple awaiters suspended
```

Fig. 4. Test conditions for the clock.

```
#begin
    #test C2 C5
    #tick                  // T1
        #thread
            #excMonitor m.await(1); #end
            #valueCheck time() # 2 #end
        #end
    #end
    #tick                  // T2
        #thread
            #valueCheck m.time() # 0 #end
            #excMonitor m.tick(); #end
            #valueCheck m.time() # 1 #end
            #valueCheck time() # 2 #end
        #end
    #end
#end
```

Fig. 5. A test sequence for the clock monitor.

second tick block completes, ConAn's time has advanced to 2. However, the CUT'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.

5.4 Readers-Writers

The readers and writers problem [43] is an abstraction of the problem of separate threads accessing a shared resource such as a file or database. A reader thread 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 ConAn to the readers and writers problem proved relatively straightforward with eight sequences and a total of 42 monitor calls to test 15 conditions. The construction of each test sequence was straightforward, compared with the nontrivial exercise of creating one long sequence consisting of 31 monitor calls [25]. Using multiple shorter test sequences has greatly simplified the selection of test cases to cover the test conditions.

5.5 Testing Mutant Reader-Writer Monitors

To further evaluate ConAn, we used the same seven faulty mutant versions of the readers-writers monitor implementation that were used in [25]. The mutants were created by modifying the code to provide an incorrect implementation. For example, in one mutant, the `notifyAll()` method, which wakes all threads suspended, was replaced with the `notify()` method, which wakes only one suspended thread. In another mutant, the `while` loop used to control monitor re-entry of suspended threads, was replaced with an `if` statement. These mutations are typical of errors made in concurrent Java programs. By implementing the new `time` function and thread suspension detection, we were able to determine the time when a thread completes a call and were successful in detecting all seven faulty mutants. In [25], only three faulty mutants were detected because there was no way to detect when a thread completed a call.

5.6 Testing Student Assignments

ConAn was used to help with the assessment of 83 student assignments in a third-year concurrent programming course. As part of the assignment, the students were asked to implement a bounded buffer controlled by semaphores. The students were not allowed to use the synchronized keyword in the bounded buffer implementation, but were required to use the provided semaphore class. This meant that the students had to use semaphores to guarantee mutually exclusive access when updating the buffer.

We wrote a ConAn test script containing five test sequences similar to the ones described in this paper. A sixth sequence was used to test the provision of mutual exclusion. In this test sequence, we unleashed 10 producer threads and 10 consumer threads all at the same time. Each of the producer threads put 100 numbers in the buffers and each of the consumer threads took out 100 numbers. Clearly, the order in which this happens is impossible to predict and the only check that was performed is that the same 1,000 numbers that were put into the buffer were retrieved.

The assumption was that, if the buffer was not protected against data races by enforcing mutually exclusive access by producer and consumer threads, then the check would fail at some stage in the sequence. To check this assumption, we modified our own solution by eliminating the semaphores for mutually exclusive access and ran the ConAn script 100 times. It detected the error 69 times, which was deemed sufficient for our purposes (although it does mean we might miss a faulty student implementation). In the actual test run, the test sequence detected an error in 24 out of the 83 assignments.

5.7 Textbook Examples: Components 4 to 14

Components 4 to 14 are all the monitors contained in [43]. There were no errors found in the latest implementations of these monitors. However, earlier versions of the text (and the currently available implementations for download) contain errors in `BufferImpl` and `ReaderWriterPriority`. The `BufferImpl` monitor incorrectly uses `notify` rather than `notifyAll`. This produces erroneous behavior in the form of permanently suspended threads and was detected by ConAn as shown in the following test output:

```
Sequence 6: liveness error detected at time: 3.
  Thread: producer2
Sequence 6: liveness error detected at time: 5.
  Thread: consumer4
***** Test cases: 50
***** Value errors: 0
***** Exception errors: 0
***** Liveness errors: 2
```

`ReaderWriterPriority` also incorrectly used a `notify` in the `releaseRead` method. This was difficult to detect due to the nondeterministic waking of suspended threads and led us to attempt to force a test case failure by utilizing thread priorities to modify the wake-up times of sleeping threads. This proved very difficult using Sun's JVM (build "1.3.1_01") for the Windows98 platform. However, we were able to force the error by using a JVM (build "chapman:10/12/12-23:12") on the Linux operating system (Redhat 4.2). Although thread priorities exist in Java and many references state that the JVM will always select one of the highest priority threads for scheduling [22], [24], [39], this is currently not guaranteed by the Java language or virtual machine specifications [23], [40]. Priorities are only hints to the scheduler [52, p. 227].

Generally, ConAn is effective at dealing with the nondeterminism of waking threads. 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 or more possible wake-up times for each thread. This was achieved by supplying a list of valid wakeup times to the wake up check. If the check passed, the wakeup time was removed from the list. The next woken thread would then check against the remaining values in the list.

5.8 Industry Examples: Components 15 to 20

`BlastServer` is used in a middleware application [41] and is a variation of the producer-consumer monitor. No faults were found in the implementation of this monitor. Components 16 to 20 are used in a commercial e-commerce application. ConAn found faults in three of the six components tested.

`BlockingTerminalPool` keeps a list of resources (called terminals) that can be used by a client of the class. There is a maximum number of terminals that can be used at any time. The `BlockingTerminalPool` serves out terminals to clients up to a maximum value and, when reached, blocks the calling thread until more terminals are available. One of the methods to retrieve a list of terminals allocates an array with the number of terminals contained in the list. It then proceeds to iterate over the elements. Another method removes terminals from the list. Both methods are unsynchronized. The following sequence illustrates the fault found by ConAn:

- Thread T_1 allocates an array of size s.
- Thread T_2 removes a terminal from the list, the size of the list becoming $s - 1$.
- Thread T_1 iterates over the list to build the array and only inserts $s - 1$ items into the array of size s. The client of the call is returned an array with one less element than expected.

The other fault found is similar in nature but involves a method that adds items into the list and is illustrated by the following sequence of events:

- Thread T_1 allocates an array of size s.
- Thread T_2 adds a terminal to the list, the size of the list becoming $s + 1$.
- Thread T_1 iterates over the list to build the array and attempts to insert $s + 1$ items into an array of size s, resulting in a runtime exception.

`JDBCConnectionPool` is a class for pooling database connections. It incorrectly re-entered the wait state when a timeout occurred on the `wait` statement. The statement that would throw an exception on timeout was placed in the wrong part of the program and, hence, was never reached in the intended circumstances.

`RecordInputStream` is a bounded circular buffer for handling the reading and writing of byte streams. ConAn detected three significant and distinct errors. These consisted of two liveness errors and a race condition. The race condition was caused by the nonatomic actions of checking the size of the buffer prior to removal of an item. Without synchronization, a number of threads would find the buffer nonempty. When the item is removed by one of the threads, the others fail to retrieve the item and throw an exception.

The liveness errors were related to incorrect processing logic for *wait* and *notify* events. Rather than notifying all threads, only one thread was being notified. Once notified, a Boolean flag was being set to prevent further threads from being notified, thus causing the remaining suspended threads to be permanently suspended.

6 Conclusion

The nondeterministic nature of concurrent programs means that conventional testing methods are inadequate. Our strategy for testing such programs is to obtain adequate controllability over the interactions between multiple threads.

The test method is derived from an existing method [7] that tests Concurrent Pascal monitors. The original method consists of four steps. We have simplified the method to three steps: identifying preconditions, constructing a sequence of calls, and execution and comparison. In earlier work [25], 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 have provided further tool support through ConAn, which automates the generation of the test driver from the test sequences. In addition, we implemented a time function in the clock, to check when calls complete and to provide the ability to detect liveness errors. We also improved the method by using multiple shorter test sequences, rather than one long sequence. We discussed the application of ConAn and the method to a range of monitors and extended its application to other concurrent components. ConAn detected faults in five of the 20 components tested, including three from industry.

Our experience has shown that ConAn test suites are good at detecting both liveness and safety errors in concurrent programs. Deadlock and other faults that cause thread suspension are often detected by the test suites. There are still some cases that are difficult to detect due to the nondeterministic waking of threads from the wait set. In fact, we have found only one case to date that has gone undetected due to nondeterministic waking of threads and this was related to a specific JVM implementation. In other cases, ConAn handles nondeterministic thread notification by checking against a number of valid wake up times.

Although ConAn has been successful at detecting race conditions and other faults related to synchronization of critical sections, we cannot guarantee that it will be successful in detecting all such conditions. We plan future work in this area in two directions:

1. We plan to develop a model of common faults/failures in concurrent Java components and then to

investigate the types of faults/failures that ConAn is good at detecting and what test case selection strategies are best to uncover these types of faults/failures. Where ConAn cannot reliably uncover certain types of faults/failures, such as race conditions, we plan to investigate other suitable techniques and tools, such as code instrumentation, model checkers, and the Java JDI and JVMDI APIs for querying runtime state information.

2. We plan to investigate the use of thread priorities in JVMs that implement priority scheduling as expected. This will solve the nondeterministic thread wake-up problem and should eliminate the need to rely on a fixed tick time interval.

Other future work involves the testing of particular types of components, such as Enterprise JavaBeans. Here, we should be able to exploit the commonalities in these components to come up with patterns or strategies for both test case selection and test case execution (for example, by generating parts of a test script automatically).

Appendix

ConAn Syntax and Semantics

Fig. 6 shows the general structure of a ConAn test script. Each section of the test script is described below.

- Tick block: Each tick block, delimited by `#tick` and `#end`, represents a unit of time (or *tick* of the clock). Each tick has a duration of *tickTime* milliseconds as defined by the `#ticktime` statement. It is assumed that any statement executing within a tick block will complete before *tickTime* milliseconds has passed.
- Thread block: Each thread block begins with the `#thread` statement that identifies the thread that will execute the enclosed Java code by the thread identifier *thread-id* (specified within angle brackets). If the thread identifier is omitted, ConAn generates a unique identifier for the thread block. Thread blocks within different tick blocks can use the same identifier, in which case the code associated with those thread blocks is executed by the same thread. Each thread identifier may appear only once in each tick block.

  A thread with identifier id executes code for each tick block T that it 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. Any Java code can be entered between the `#thread` and `#end` statements.
- `#ticktime` *tickTime*: sets the number of milliseconds allocated for each *tick block*. The default value of 1,000 milliseconds can be overidden by using the optional `#ticktime` command.
- `#cut` *cut-id class-constructor*: The *class-constructor* is the constructor of the component under test (CUT). Parameters may be supplied to the constructor in the usual way. If no parameters are required to call the constructor, the no-arg parentheses are not required. ConAn will add these during driver generation. The *cut-id* is any valid Java variable name and is used to refer to an instance of the CUT. This identifier can then be used in the test sequences to reference the CUT instance. A new CUT instance is created for each test sequence.

```
#ticktime tickTime

#cut cut-id class-constructor

#conditions
    condition-id condition-description
    ...
#end

#top
    Java code // will be inserted at top of driver
#end

#shared ⟨instance-id⟩
    Java code // global shared variables and setup code
#end

#begin // first test sequence
    #test condition-id ...
    #shared ⟨instance-id⟩
        Java code // shared variables and setup code
    #end
    #tick // first tick block
        #thread ⟨thread-id⟩
            Java code // code for this thread
        #end
        #thread ⟨thread-id⟩
            Java code // code for this thread
        #end
        ...
    #end
    #tick // second tick block
        ...
    #end
    ...
#end

#begin // second test sequence
    ...
#end
```

Fig. 6. Structure of a ConAn test script.

- *condition-id condition-description*: Conditions are listed in the conditions block, delimited by `#conditions` and `#end`. 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 `#test` 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 `#test` list of any test sequence are reported by ConAn as potential problems. However, the use of conditions in a test script is optional.
- `#top`: Code to be inserted at the top of the driver program is included in the `#top` block. This is useful for specifying classes to be imported for use within the test script.

- Test sequence: A test script consists of one or more test sequences, delimited by `#begin` and `#end`. Each test sequence consists of a number of tick blocks representing *ticks* of length *tickTime* milliseconds. A test sequence completes after $n + 1$ ticks have passed, where n is the number of tick blocks in the sequence. At this time, the driver checks to make sure there are no suspended threads for that test sequence.
- `#test` *condition-id*: For each test sequence, a number of test 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.
- `#shared` $\langle$*instance-id*$\rangle$: Shared variables, methods, and setup and teardown code, delimited by `#shared` and `#end`, can be inserted at the beginning of a test sequence. A shared block defines a class that, when instantiated, creates a shared object that is passed to each thread in a test sequence. This means that every variable and method defined in the shared block is available to all threads in a test sequence, referenced by *instance-id.name*.

  Shared blocks may be defined either once outside the test sequences (global shared block) or within a test sequence (test sequence shared block). Any variables and methods defined within the global shared block are available to all test sequences. Variables and methods defined in a test sequence inherit from the global shared block, so all variables and methods are accessible by the test sequence shared block. Moreover, methods in the test sequence shared block can override global shared methods as desired.

  The base class, which the shared blocks extend, provides two methods used by generated drivers, `setup()` and `teardown()`. The setup (teardown) method is executed immediately before (after) each test sequence. The tester can override these methods in the shared blocks to provide their own setup and teardown functionality.

REFERENCES

[1] G. Andrews, *Concurrent Programming: Principles and Practice.* Addison Wesley, 1991.

[2] C. Artho and A. Biere, "Applying Static Analysis to Large-Scale, Multi-Threaded Java Programs," *Proc. 2001 Australian Software Eng. Conf.,* pp. 68-75, 2001.

[3] A. Bechini, J. Cutajar, C. Bochmann, and A. Petrenko, "A Tool for Testing of Parallel and Distributed Programs in Message Passing Environments," *Proc. Ninth Mediterranean Electrotechnical Conf. 1998 (MELECONE98),* vol. 2, pp. 1308-1312, 1998.

[4] A. Bechini and K.-C. Tai, "Design of a Toolset for Dynamic Analysis of Concurrent Java Programs," *Proc. Sixth Int'l Workshop Program Comprehension,* pp. 190-197, 1998.

[5] K. Beck, *Extreme Programming Explained.* Addison Wesley, 2000.

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

[7] P. Brinch Hansen, "Reproducible Testing of Monitors," *Software–Practice and Experience,* vol. 8, pp. 721-729, 1978.

[8] X. Cai and J. Chen, "Control of Nondeterminism in Testing Distributed Multithreaded Programs," *Proc. First Asia-Pacific Conf. Quality Software,* 2000.

[9] R. Carver and K.-C. Tai, "Deterministic Execution Testing of Concurrent Ada Programs," *Conf. Proc. Ada Technology in Context: Application, Development, and Deployment,* pp. 528-544, 1989.

[10] R.H. Carver and K.-C. Tai, "Replay and Testing for Concurrent Programs," *IEEE Software,* vol. 8, no. 2, pp. 66-74, 1991.

[11] R.H. Carver and K.-C. Tai, "Use of Sequencing Constraints for Specification-Based Testing of Concurrent Programs," *IEEE Trans. Software Eng.,* vol. 24, no. 6, pp. 471-490, 1998.

[12] J. Choi and H. Srinivasan, "Deterministic Replay of Java Multithreaded Applications," *Proc. SIGMETRICS Symp. Parallel and Distributed Tools,* pp. 48-59, 1998.

[13] E.M. Clarke, E.A. Emerson, and A.P. Sistla, "Automatic Verification of Finite-State Concurrent Systems Using Temporal Logic Specifications," *ACM Trans. Programming Languages and Systems,* vol. 8, no. 2, pp. 244-263, Apr. 1986.

[14] C. Corbett, M. Dwyer, J. Hatcliff, S. Laubach, C. Pasareanu, Robby, and H. Zheng, "Bandera: Extracting Finite-State Models from Java Source Code," *Proc. 22nd Int'l Conf. Software Eng.,* 2000.

[15] J.C. Corbett, "Constructing Compact Models of Concurrent Java Programs," *Proc. ACM SIGSOFT Int'l Symp. Software Testing and Analysis,* pp. 1-10, 1998.

[16] N. Daley, D.M. Hoffman, and P.A. Strooper, "A Framework for Table Driven Testing of Java Classes," *Software–Practice and Experience,* vol. 32, pp. 465-493, 2002.

[17] C. Demartini and R. Sisto, "Static Analysis of Java Multithreaded and Distributed Applications," *Proc. Int'l Symp. Software Eng. for Parallel and Distributed Systems,* pp. 215-222, 1998.

[18] O. Edelstein, E. Farchi, Y. Nir, G. Ratsaby, and S. Ur, "Multithreaded Java Program Test Generation," *IBM Systems J.,* vol. 41, no. 1, pp. 111-125, 2002.

[19] M. Fowler, *Refactoring: Improving the Design of Existing Code.* Addison Wesley, 1999.

[20] H. Gaifman, M. Maher, and E. Shapiro, "Replay, Recovery, Replication, and Snapshots of Nondeterministic Concurrent Programs," *Proc. 10th Ann. ACM Symp. Principles of Distributed Computing,* pp. 241-255, 1991.

[21] E. Gamma and K. Beck, "JUnit Testing Framework," available online at http://www.junit.org, 2002.

[22] J. Gosling and K. Arnold, *The Java Programming Language,* second ed. Addison Wesley, 1998.

[23] J. Gosling, B. Joy, G. Steele, and G. Bracha, *The Java Language Specification,* second ed. Addison Wesley, online at http://java.sun.com/docs/books/jls/index.html, 2000.

[24] S.J. Hartley, *Concurrent Programming: The Java Programming Language.* p. 55, Oxford Univ. Press, 1998.

[25] C. Harvey and P. Strooper, "Testing Java Monitors through Deterministic Execution," *Proc. 2001 Australian Software Eng. Conf.,* pp. 61-67, 2001.

[26] J. Hatcliff and M. Dwyer, "Using the Bandera Tool Set to Model-Check Properties of Concurrent Java Software," *Proc. 12th Int'l Conf. Concurrency Theory (CONCUR'01),* pp. 39-58, 2001.

[27] K. Havelund, "Java PathFinder, a Translator from Java to Promela," *Proc. Fifth and Sixth SPIN Workshops,* 1999.

[28] K. Havelund, "Using Runtime Analysis to Guide Model Checking of Java Programs," *Proc. Seventh SPIN Workshop,* pp. 245-264, 2000.

[29] K. Havelund and G. Rosu, "Monitoring Java Programs with Java PathExplorer," *Proc. First Int'l Workshop Runtime Verification (RV'01),* vol. 55 of Electronic Notes in Theoretical Computer Science, K. Havelund and G. Rosu, eds., pp. 97-114, 2001.

[30] K. Havelund and G. Rosu, "Monitoring Programs Using Rewriting," *Proc. 16th Int'l Conf. Automated Software Eng.,* pp. 135-143, 2001.

[31] C.A.R. Hoare, *Communicating Sequential Processes.* Prentice Hall, 1985.

[32] D.M. Hoffman and P.A. Strooper, "Techniques and Tools for Java API Testing," *Proc. 2000 Australian Software Eng. Conf.,* pp. 235-245, 2000.

[33] E. Itoh, Z. Furukawa, and K. Ushijima, "A Prototype of a Concurrent Behavior Monitoring Tool for Teesting of Concurrent Programs," *Proc. Third Asia-Pacific Software Eng. Conf.,* 1996.

[34] R. Jeffries, "Extreme Testing," *Software Testing and Quality Eng.,* pp. 23-26, Mar. 1999.

[35] T. Katayama, Z. Furukawa, and K. Ushijima, "Design and Implementation of Test-Case Generation for Concurrent Programs," *Proc. 1998 Asia-Pacific Software Eng. Conf.,* pp. 262-269, 1998.

[36] T. Katayama, E. Itoh, and Z. Furukawa, "Test-Case Generation for Concurrent Programs with the Testing Criteria Using Interaction Sequences," *Proc. 2000 Asia-Pacific Software Eng. Conf.*, pp. 590-597, 2000.

[37] R. Konuru, H. Srinivasan, and J. Choi, "Deterministic Replay of Distributed Java Applications," *Proc. 14th Int'l Parallel and Distributed Processing Symp.*, pp. 219-227, 2000.

[38] P.V. Koppol and K.-C. Tai, "An Incremental Approach to Structural Testing of Concurrent Programs," *Proc. 1996 Int'l Symp. Software Testing and Analysis*, pp. 14-23, 1996.

[39] D. Lea, *Concurrent Programming in Java.* pp. 210-211, Addison Wesley, 1997.

[40] T. Lindholm and F. Yellin, *The Java Virtual Machine Specification*, second ed. Addison Wesley, online at http://java.sun.com/docs/books/vmspec/index.html, 1999.

[41] B. Long and P. Strooper, "A Case Study in Testing Distributed Systems," *Proc. Third Int'l Symp. Distributed Objects and Applications*, pp. 20-29, 2001.

[42] D. Long and L.A. Clarke, "Data Flow Analysis of Concurrent Systems that Use the Rendezvous Model of Synchronisation," *Proc. Symp. Software Testing, Analysis and Verification (TAV4)*, pp. 21-35, 1991.

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

[44] C.E. McDowell and D.P. Helmbold, "Debugging Concurrent Programs," *ACM Computing Surveys*, vol. 21, no. 4, pp. 593-622, 1989.

[45] M. McMillan, "Symbolic Model Checking," PhD thesis, Carnegie Mellon Univ., 1992.

[46] R. Milner, *Communication and Concurrency.* Prentice Hall, 1989.

[47] G. Naumovich, G. Avrunin, and L. Clarke, "Data Flow Analysis for Checking Properties of Concurrent Java Programs," *Proc. 1999 Int'l Conf. Software Eng.*, pp. 399-410, 1999.

[48] M. Nygard and T. Karsjens, "Test Infect Your Enterprise JavaBeans," *JavaWorld J.*, May 2000.

[49] E.H. Paik, Y.S. Chung, B.S. Lee, and C.-W. Yoo, "A Concurrent Program Debugging Environment Using Real-Time Replay," *Proc. 1997 Conf. Parallel and Distributed Systems*, 1997.

[50] D. Park, U. Stern, J. Skakkebaek, and D. Dill, "Java Model Checking," *Proc. 15th Int'l Conf. Automated Software Eng.*, 2000.

[51] S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, and T. Anderson, "Eraser: A Dynamic Data Race Detector for Multithreaded Programs," *ACM Trans. Computer Systems*, vol. 15, no. 4, pp. 391-411, Nov. 1997.

[52] C. Szyperski, *Component Software: Beyond Object-Oriented Programming.* Addison Wesley, 1998.

[53] C. Szyperski and C. Pfister, "Special Issues in Object-Oriented Programming-ECOOP 96 Workshop Reader," *Workshop on Component-Oriented Programming, Summary*, M. Muhlhauser, ed., 1997.

[54] K.-C. Tai, R.H. Carver, and E. Obaid, "Debugging Concurrent Ada Programs by Deterministic Execution," *IEEE Trans. Software Eng.*, vol. 17, no. 1, pp. 45-62, 1991.

[55] R.N. Taylor, "A General-Purpose Algorithm for Analyzing Concurrent Programs," *Comm. ACM*, vol. 26, no. 5, pp. 362-376, 1983.

[56] R.N. Taylor, D.L. Levine, and C.D. Kelly, "Structural Testing of Concurrent Programs," *IEEE Trans. Software Eng.*, vol. 18, no. 3, pp. 206-215, 1992.

[57] W. Visser, K. Havelund, G. Brat, and S. Park, "Model Checking Programs," *Proc. 15th Int'l Conf. Automated Software Eng.*, 2000.

[58] M. Young and R.N. Taylor, "Combining Static Concurrency Analysis with Symbolic Execution," *IEEE Trans. Software Eng.*, vol. 14, no. 10, pp. 1499-1511, Oct. 1988.

Brad Long received the BSc degree in computer science in 1988 from the University of Queensland, and the MBA degree in 1999 from the University of Southern Queensland. He is currently working on a PhD degree at the University of Queensland on the topic of testing concurrent software components. He is a project leader at the Australian Development Centre, Oracle Corporation, Brisbane, Australia. Since 1988, he has worked as a software engineer for a number of national and international firms including Mincom Pty Ltd., Fujitsu Ltd., and the Oracle Corporation. His research interests include software engineering, especially software verification and testing and distributed systems. He is a member of the IEEE Computer Society.

Daniel Hoffman received the BA degree in mathematics from the State University of New York, Binghamton, in 1974, and the MS and PhD degrees in computer science in 1981 and 1984, from the University of North Carolina, Chapel Hill. From 1974 to 1979, he worked as a commercial programmer/analyst. He is currently an associate professor of computer science at the University of Victoria (British Columbia, Canada). His research emphasizes the industrial application of software documentation, inspection, and testing. He spent the 1992-1993 year on leave at Tandem Computers, Inc. working on software inspection and automated class testing, and the 1998-1999 year on sabbatical at Bell Laboratories, Lucent Technologies, doing research in software product lines. He is a member of the IEEE Computer Society and the IEEE.

Paul Strooper received the BMath and MMath degrees in computer science in 1986 and 1988 from the University of Waterloo, and the PhD degree in computer science in 1990 from the University of Victoria. He is a senior lecturer in the School of Information Technology and Electrical Engineering at The University of Queensland. From 1990 to 1992, he worked as a research associate for the Institute for Robotics and Intelligent Systems at the University of Victoria. His research interests include software engineering, especially software specification, verification, and testing, and logic programming, especially program transformation and refinement. He is an associate academic of the Software Verification Research Centre and regularly teaches industrial training courses for the SVRC. He is a member of the IEEE Computer Society and the IEEE.