A Classification of Concurrency Failures in Java Components

Brad Long[*] Paul Strooper

Software Verification Research Centre,
School of Information Technology and Elec. Eng.,
The University of Queensland,
Brisbane, Qld 4072, Australia.
email: {brad, pstroop}@itee.uq.edu.au

Abstract

The Java programming language supports concurrency. Concurrent programs are hard to test due to their inherent non-determinism. This paper presents a classification of concurrency failures that is based on a model of Java concurrency. The model and failure classification is used to justify coverage of synchronization primitives of concurrent components. This is achieved by constructing concurrency flow graphs for each method call. A producer-consumer monitor is used to demonstrate how the approach can be used to measure coverage of concurrency primitives and thereby assist in determining test sequences for deterministic execution.

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. Implementing and testing concurrent programs is difficult due to the inherent non-determinism in these programs. That is, if we run a concurrent program twice with the same input, it is not guaranteed to return the same output both times.

By concentrating on components we do not need to be concerned with the number of threads that are executing in the entire system, because we assume the component can be accessed by any number of threads at a time. That is, we test a component under the assumption of multiple thread access. Following Szyperski [26], 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.

[*] also Australian Development Centre, Oracle Corporation, 300 Ann St, Brisbane, Qld 4000, Australia.

Unit testing is becoming an increasingly popular approach for assisting the development of high quality software [4, 17]. Whilst unit testing tools exist for sequential programs, there are currently no tools for the systematic unit testing of concurrent software components.

Our first step for systematic testing is to develop a model of Java concurrency. Petri-nets [23] are used to represent the model in a graphical manner. The model is used to discuss important synchronization points in concurrent Java components. A classification of concurrency failures is presented along with the model. From the classification and model, synchronization flow graphs are created for each method in the concurrent program. The synchronization flow graphs may then be used to assist the construction of test sequences that cover the arcs of the graphs. The test sequences can be used to construct test drivers or as input to dynamic analysis testing tools [19, 20].

We review related work in Section 2. In Section 3 we provide an overview of Java concurrency constructs. The model of Java concurrency is presented in Section 4. Section 5 explains the classification of concurrency failures. The application of the model to test case selection is presented in Section 6.

2 Related Work

Static analysis of concurrent programs involves the analysis of a program without requiring execution. Typically this involves the generation and analysis of (partial) models of the states and transitions of a program [18, 21, 22, 27]. 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.

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. This approach facilitates formal analysis and mechanical checking of software systems, 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. More recently, models have been successfully generated automatically from the program source or object code. Approaches based on model checking include Bandera [14], JPF [15, 28], and Jlint [2].

Some tools combine static and dynamic analysis. For example, JPF's runtime analysis utilizes the LockTree and Eraser [24] algorithms for detecting potential deadlocks and race conditions. In [7], the static analysis phase collects information to allow the more accurate dynamic phase to execute efficiently. Other recent work [8] focusses on parameterized verification to handle infinite-state abstraction models of concurrent Java programs.

Deterministic testing of concurrent programs [3, 5, 9, 10, 19, 20, 25] 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 run-time 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.

In this paper, we develop a generic model for a concurrent thread accessing an object lock. We then use the model to classify failures in concurrent components. A graphical representation of a concurrent component is then constructed, and the graph is used to guide test sequence construction to ensure coverage of concurrency primitives.

3 Overview of Concurrency in Java

Some basic Java concurrency constructs are reviewed in this section before proceeding with the formal specification. Thread creation, `join`, `sleep`, and `interrupt` are not discussed since these are not typically found in concurrent components themselves, but in the multithreaded programs that use these components. The methods `suspend`, `resume` and `stop` are also not discussed because they are deprecated and their use is discouraged [16].

3.1 Mutual Exclusion and Object Locking

In the Java programming language [11, 12], mutual exclusion is achieved by a thread locking an object. Two threads cannot lock the same object at the same time, thus providing mutual exclusion. A thread that cannot access a synchronized block because the object is locked by another thread is *blocked*. In Java there are two ways of locking an object.

1. Explicitly call a synchronized block.

   ```
   synchronized (anObject) {
       ...
   }
   ```

   The Java code above, locks the object `anObject`. The lock is released when the executing code leaves the synchronized block. If another thread is already executing code within the synchronized block, the requesting thread will be blocked until the thread holding the lock leaves the synchronized block.

2. Synchronize a method.

   ```
   public synchronized void aMethod() {
       ...
   }
   ```

   The Java code above, which synchronizes a method, is the same as locking the `this` object in a synchronized block. The following code provides identical behaviour:

   ```
   public void aMethod() {
       synchronized (this) {
           ...
       }
   }
   ```

A thread can lock more than one object. For example, the thread executing the following Java code locks the two objects `object1` and `object2`. Both locks are held whilst in the inner-most synchronized block. As each block is exited, the associated lock is released.

```
synchronized (object1) {
    ...
    synchronized (object2) {
        ...
    }
}
```

3.2 Waiting and Notification

Threads are suspended by calling the Java `wait` method. This causes the lock on the object to be released, allowing other threads to obtain a lock on the object. Suspended threads remain dormant until woken. As an example, a particular implementation of the producer-consumer monitor provides two methods, `put` and `get`. The `put` method places an item into the buffer and the `get` method retrieves an item from the buffer. A thread will be suspended via the `wait` statement if it calls `get` whilst there are no items in the buffer.

```
public synchronized Item get() {
    while (buffer.size() == 0) {
        wait();
    }
    ...
}
```

A thread calling `notify` will cause the run-time scheduler, managed by the Java Virtual Machine (JVM), to arbitrarily select a waiting thread to be woken. The selected thread will then attempt to regain the object lock for re-entry to the synchronized block immediately after the call to `wait`. For the producer-consumer monitor, the `put` call places an item into the buffer and then notifies a waiting thread. There is also a method `notifyAll` that wakes all waiting threads on the object.

```
public synchronized void put(Item item) {
    ...
    buffer.add(item);
    notify();
}
```

4 A Model of Java Concurrency

Figure 1 models the states of a single thread with respect to a synchronized object by using a petri-net diagram [23]. This representation has been chosen to highlight two issues: 1) the change in state of a thread when concurrent constructs are encountered in a multithreaded program, and 2) the effect that availability of the object lock has on a thread's state. The diagram contains *markers* (shown as dots) and two types of nodes: circles (called *places*) and bars (called *transitions*). These places and transitions are connected by directed arcs from places to transitions and from transitions to places. A transition can *fire* if all incoming arcs originate from places containing markers. When a transition fires, each outgoing arc deposits a marker in its destination place. It is not proposed that petri-nets are created for specific applications. The petri-net representation is used to model possible states and transitions of a thread at any point in time.

Places A to D represent the current state of a thread. Place E represents the availability of an object lock. More specifically, the marker in place A represents a thread executing outside a synchronized block. A marker in place B represents a thread requesting entry to a critical section. A marker in place C represents a thread executing in a critical section. A marker in place D represents a thread in the *wait* state. The marker in place E means that an object lock is available.

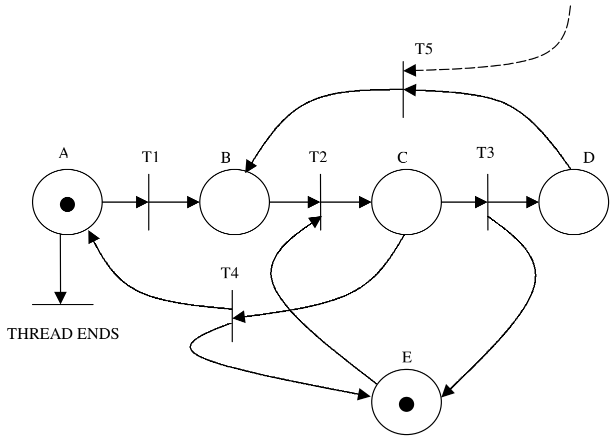

Figure 1. Petri-net model of concurrency

Transition T1: requesting an object lock

Transition T1 is fired by a thread entering a `synchronized` block. A marker exists in place A, therefore transition T1 can fire causing the marker to move to B. Place B represents a thread requesting an object lock.

Transition T2: locking an object

Transition T2 is fired by the JVM serving the requesting thread an object lock. If an object lock is available, that is, if a marker exists in place E, the marker can move to C. Place C represents a thread executing in a critical section with the object lock. If no lock is available, the thread is blocked in B.

Transition T3: waiting on an object

Transition T3 represents a thread entering the wait state. This occurs when the code calls the `wait` method, which also releases the object lock, hence the arc to place E. From C, a marker is moved to both D and E.

Transition T4: releasing an object lock

Transition T4 represents a thread leaving a synchronized block. When this occurs, a marker is placed in both A and E. This transition causes the lock on the object to be released.

Transition T5: thread notification

Transition T5 represents a waiting thread waking up. When this occurs, the marker moves to B to reacquire the object lock it was waiting on. The incoming dashed arc at

T5 represents another thread notifying the waiting thread. This has the obvious implication that a thread in the wait state cannot wake itself.

5 A Classification of Concurrency Failures

Following techniques of hazard/safety analysis, failure conditions are identified for each of the transitions. This approach is taken for completeness, to ensure all failures are identified and classified. Using a HAZOP style of analysis [6], we analyze each transition for two deviations, 1) failure to fire the transition, and 2) erroneous firing of the transition. The correct transition firings plus the two deviations form a complete set of transition firings. Table 1 contains the result of the analysis.

The columns of the table are as follows:

- Transition: the name of the transition under analysis (see Figure 1).
- Failure: a categorization of the failure. Two classifications, *failure to fire* and *erroneous firing*, are used.
- Cause: a brief description of possible causes of the failure.
- Conditions: the conditions under which the failure can occur.
- Consequences: the consequences of the failure.
- Testing Notes: any notes relating to testing implications. Generally a method or approach for detecting the failure is listed. *Check call completion time* refers to a technique used in previous work [19, 20]. This technique uses deterministic execution to allow a tester to specify sequences of method calls. 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` call allows a tester to ensure each thread wakes up at a certain time or between a range of times.

5.1 Transition T1 failures

5.1.1 Failure to fire T1 (FF-T1)

For this failure to occur, there must be shared resources that are potentially accessed by multiple threads. Failing to fire this transition means the thread does not enter a synchronized block for mutually exclusive access to any shared resources. Detecting this failure requires the ability to detect multiple threads accessing a shared resource.

5.1.2 Erroneous firing of T1 (EF-T1)

This occurs when a thread enters a synchronized block when it is not required to. This is not necessarily a serious problem, since it does not cause a failure as such, it simply introduces inefficiency into the component.

5.2 Transition T2 failures

5.2.1 Failure to fire T2 (FF-T2)

As mentioned in Table 1, this failure can occur in two ways. One way is when the thread can never get hold of an object lock because the lock is permanently held by another thread (see failure FF-T4). This can occur if the thread holding the lock is waiting for blocking input and no input is ever received, or is deadlocked.

The other way this failure can occur is if the JVM is not fair when handing out locks. If there is high contention and there is always more than one thread requesting a lock, it is possible that one thread is never selected to receive a lock. This is very difficult to detect as it depends on timing of thread requests. Since the Java virtual machine is not required to be fair, this could be a potential problem where multiple waiting threads are involved. Further details of a particular JVM implementation would be required to determine if the potential problem has been eliminated, for example, if an implementation used a first-in first-out queue for waiting threads.

5.2.2 Erroneous firing of T2 (EF-T2)

We assume the JVM is implemented correctly and therefore do not analyze this failure any further.

5.3 Transition T3 failures

5.3.1 Failure to fire T3 (FF-T3)

To detect whether a thread fails to enter a waiting state, threads can be forced to execute a sequence of calls that require them to wait. A thread that fails to enter the waiting state will complete before it should.

5.3.2 Erroneous firing of T3 (EF-T3)

To detect whether a thread erroneously enters a wait state, a similar technique can be used for the *failure to fire* case. That is, force threads to execute a sequence that is not expected to suspend threads. Then check the completion times

Transition	Failure	Cause	Conditions	Consequences	Testing Notes
T1	Failure to fire T1	Thread does not access a synchronized block when required	Two or more threads access a shared resource	Interference (also known as a race condition or data race)	Static analysis / model checking (often combined with dynamic analysis)
	Erroneous firing of T1	Program logic accesses critical section	No more than one thread accesses shared resources. The thread is not required to wait or notify other threads.	Unnecessary synchronization	Static analysis / model checking (often combined with dynamic analysis)
T2	Failure to fire T2	The object lock to be acquired has been acquired by another thread	Another thread has acquired the lock being acquired by this thread. This can occur in 2 ways: 1) one thread continuously holds the lock, or 2) one or more threads repeatedly acquire the lock being requested by this thread.	The thread is permanently suspended	Static and dynamic analysis
	Erroneous firing of T2	Not applicable			
T3	Failure to fire T3	No call to `wait` is made	Thread is required to make a call to `wait`	Program code may erroneously execute in a critical section, or leave critical section prematurely.	Check completion time of call
	Erroneous firing of T3	Program logic makes an erroneous call to `wait`	A call to `wait` is not desired	A thread may suspend indefinitely if no other thread exists to notify it. The object lock is released.	Check completion time of call
T4	Failure to fire T4	The thread never releases object lock.	Thread is either in endless loop, waiting for blocking input (which is never received), or acquiring an additional lock which is locked by another thread	Thread never completes. Other threads may be blocked if they are waiting for the lock.	Check completion time of call
		The thread fires T3, that is, it waits instead	None	Thread waits instead of completing and leaving the critical section.	Check completion time of call
	Erroneous firing of T4	Thread releases the object lock prematurely	None	Thread exits and subsequent statements may access shared resources.	Static analysis and completion time of call
T5	Failure to fire T5	Thread is not notified	No other thread calls notify whilst this thread is in the $wait$ state.	Thread is permanently suspended	Check completion time of call
	Erroneous firing of T5	Thread is notified before it should be	None	Thread prematurely re-enters the critical section	Check completion time of call

Table 1. Concurrency failure classification

of each call in the sequence to ensure that threads completed at the correct time and were not erroneously suspended.

5.4 Transition T4 failures

5.4.1 Failure to fire T4 (FF-T4)

Failure to fire this transition means that a thread permanently holds a lock (see FF-T2) or fires transition T3 and enters the wait state instead. If transition T3 is fired instead of T4, the thread erroneously enters the wait state (see EF-T3 for more details).

5.4.2 Erroneous firing of T4 (EF-T4)

This failure occurs when the thread releases the object lock prematurely which includes: 1) leaving a synchronized block too early, 2) reassigning a variable that was holding an object lock, and 3) firing T4 instead of T3[1].

5.5 Transition T5 failures

5.5.1 Failure to fire T5 (FF-T5)

This failure occurs when no thread calls notify whilst this thread is in the *wait* state. This includes the case where there is only one thread in the system and thus waits forever. This also occurs when a notify is called rather than a notifyAll, there is more than one thread continuously in the *wait* state, and one particular thread is never selected for notification. That is, selection of notified threads is not fair. This failure can be detected by checking completion times of component calls.

5.5.2 Erroneous firing of T5 (EF-T5)

This occurs when the thread prematurely re-enters the critical section. Detailed consequences are application specific. This can be detected by checking completion times of component calls made by the thread.

6 Applying the Model to Test Case Selection

Brinch Hansen [5] proposed an approach for testing Concurrent Pascal monitors consisting of four steps:

1. For each monitor operation, the tester identifies a set of preconditions that will cause each branch 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[2] that will execute the monitor calls as defined in the previous step.

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

In [13] 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. Tool support for the method was introduced in [19, 20]. Although the method and tool enabled us to detect erroneous implementations of monitors, it was not clear why loop coverage was chosen as a criteria for selecting test cases. This section discusses a systematic white-box approach for test case selection based on our model and classification of concurrency failures. We illustrate the approach with a producer-consumer monitor.

6.1 An example: producer-consumer monitor

The `ProducerConsumer` class shown in Figure 2 implements an asymmetric Producer-Consumer monitor, the Java equivalent of the Concurrent-Pascal program described in [5]. 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. 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. The `notifyAll` operation wakes up all suspended threads, although only one thread at a time will be allowed to access the monitor.

The model presented in Section 4 shows possible transitions made by threads in a multithreaded program. Since each transition is caused by a concurrent statement or construct, we propose that any concurrent construct used in a component should be executed in an attempt to detect any concurrency failures (as classified in Section 5). To achieve coverage of all concurrent statements, a Concurrency Flow Graph (CoFG) is constructed. Because we are focussing on constructing graphs of a concurrent component and not entire systems, constructions of CoFGs are relatively straightforward. The CoFG contains all statements that cause tran-

[1]The thread may execute and leave the critical section rather than suspending on the wait queue.

[2]These processes are scheduled by means of a clock used for testing only.

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

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

Figure 2. Producer-consumer monitor

sitions as described in our model. Each arc in the graph is a unique, although possibly overlapping, code region. To construct the CoFG for the producer-consumer monitor we identify the code regions between all pairs of concurrent statements in each method. The CoFG for the `receive` method is constructed as follows (and is represented graphically in Figure 3):

1. $start \rightarrow$ `wait`
   This is the code region between the beginning of the synchronized block[3] to the `wait` statement. The code region will be covered when the `while` statement of the `receive` method (in Figure 2) evaluates to $true$. This represents the following transition firings from our model: T1, T2, T3.

2. `wait` $\rightarrow$ `wait`
   This code region is from the one invocation of `wait` to the next. It covers the code from the end of the wait statement (the second half of the while loop) to the beginning of the next invocation of the wait statement (the first half of the while loop). The `while` condition on iteration of the loop must evaluate to $true$. This covers the transition firings T3, T5, T2, T3.

[3]In this example, the beginning of the synchronized block is also the beginning of the method.

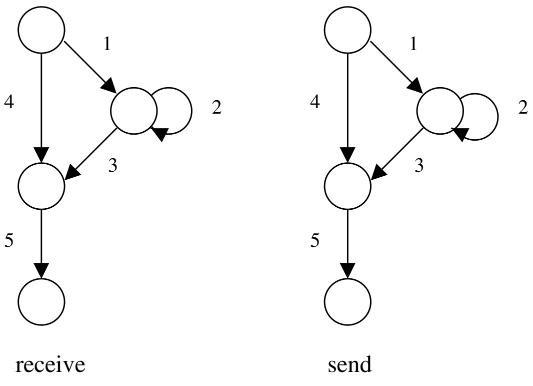

Figure 3. CoFGs for producer-consumer

3. `wait` $\rightarrow$ `notifyAll`
   This is the code region from the `wait` statement to the `notifyAll` statement. The `while` condition on iteration of the loop must evaluate to $false$. Transitions fired: T3, T4, T5.

4. $start \rightarrow$ `notifyAll`
   This code region is from the beginning of the synchronized block to the `notifyAll` statement. The `while` condition must evaluate to $false$. Transitions fired: T1, T2, T5.

5. `notifyAll` $\rightarrow end$
   The region is from `notifyAll` to the end of the synchronized block. Transitions fired: T5, T4.

The CoFG for `send` is identical to that for `receive` in this case (see Figure 3). Finally we can build test sequences that exercise arcs of the CoFGs. This involves creating a test driver that instantiates a number of threads which make calls on the synchronized methods. The sequence of calls should ensure coverage of the CoFGs. The test driver can easily be created by using the ConAn concurrency testing tool [19, 20].

7 Conclusion

The non-deterministic nature of concurrent programs means that conventional testing methods are inadequate. New techniques and tools need to be developed to allow the verification and testing of such programs. In this paper, a petri-net model of concurrency is presented and each transition is analyzed. The generic model consists of a thread interacting with an object lock. The transitions in the model represent changes in the concurrent state of a thread. From this model, a classification of concurrency failures based on transition firings is proposed. The classification is used to justify the construction of concurrency flow graphs (CoFGs) for each method in a concurrent component. Complexity is significantly reduced by focussing on concurrent components rather than entire systems. A component is tested under the assumption of multiple thread access. The producer-consumer monitor is used as an example to demonstrate the creation of concurrency flow graphs. The concurrency flow graphs can be used in the construction of test sequences for testing concurrent components to ensure coverage of concurrency primitives.

The classification for concurrency failures provides us with a motivation for a test case selection strategy using concurrency flow graphs. It potentially removes the need for white-box techniques based on previous work [19, 20]. In addition, the classification highlights the importance of checking thread completion times since this can be used in many cases to detect transition failures. By applying this technique in combination with black-box testing, we believe a superior technique to previous work [19, 20] can be devised. Future work will include 1) development of CoFGs and test sequences using this technique on a range of concurrent components, 2) a comparison of this technique with those used in earlier work, and 3) tool support for generation of CoFGs and coverage analysis during testing.

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. In *Proceedings of the 2001 Australian Software Engineering Conf.*, pages 68–75. IEEE Comp Society, 2001.

[3] A. Bechini and K-C. Tai. Design of a toolset for dynamic analysis of concurrent Java programs. In *Proceedings of the 6th International Workshop on Program Comprehension*, pages 190–197, 1998.

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

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

[6] D.J. Burns and R.M. Pitblado. A modified HAZOP methodology for safety critical assessment. In F. Redmill and T. Anderson, editors, *Directions in Safety-critical Systems: Proceedings of the Safety-critical Systems Symposium*. Springer Verlag, 1993.

[7] J-D. Choi, K. Lee, A. Loginov, R. O'Callahan, V. Sarkar, and M. Sridharan. Efficient and precise datarace detection for multithreaded object-oriented programs. In *Proceedings of the 2002 Conference on Programming Language Design and Implementation (PLDI'02)*. ACM Press, 2002.

[8] G. Delzannon, J-F. Raskin, and L. Van Begin. Towards the automated verification of multithreaded Java programs. In *Proceedings of the Int. Conf. on Tools and Algorithms for Construction and Analysis of Systems (TACAS 2002)*, pages 173–187. Springer-Verlag, 2002.

[9] O. Edelstein, E. Farchi, Y. Nir, G. Ratsaby, and S. Ur. Multithreaded Java program test generation. *IBM Systems Journal*, 41(1):111–125, 2002.

[10] E. Farchi, M. Factor, and Y. Talmor. Testing for timing-dependent and concurrency faults. In *Proceedings of the 1998 International Conference on Software Testing Analysis and Review*. Software Quality Engineering, 1998.

[11] J. Gosling and K. Arnold. *The Java Programming Language*. Addison Wesley, 2nd edition, 1998.

[12] J. Gosling, B. Joy, G. Steele, and G. Bracha. *The Java Language Specification*. Addison Wesley, 2nd edition, 2000. Also online at http://java.sun.com/docs/books/jls/index.html as at Sep 2002.

[13] C. Harvey and P. Strooper. Testing Java monitors through deterministic execution. In *Proceedings of the 2001 Australian Software Engineering Conference*, pages 61–67, 2001.

[14] J. Hatcliff and M. Dwyer. Using the Bandera tool set to model-check properties of concurrent Java software. In *Proc. 12th International Conf. on Concurrency Theory*, pages 39–58. Springer-Verlag, 2001.

[15] K. Havelund. Java PathFinder, a translator from Java to Promela. In *Proceedings of 5th and 6th SPIN Workshops*. Springer-Verlag, 1999.

[16] Sun Microsystems, Inc. Why are Thread.stop, Thread.suspend, Thread.resume and Runtime.runFinalizersOnExit deprecated? Available online at http://java.sun.com/products/jdk/1.2/docs/guide/misc/threadPrimitiveDeprecation.html.

[17] R. Jeffries. Extreme testing. *Software Testing and Quality Engineering*, pages 23–26, March 1999.

[18] 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.

[19] B. Long, D. Hoffman, and P. Strooper. A concurrency test tool for Java monitors. In *Proc. 16th International Conf. on Automated Software Engineering*, pages 421–425. IEEE Computer Society, 2001.

[20] B. Long, D. Hoffman, and P. Strooper. Tool support for testing Java monitors. Technical Report 01-21, Software Verification Research Centre, The University of Queensland, June 2001.

[21] 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.

[22] 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.

[23] J.L. Peterson. Petri nets. *ACM Computing Surveys*, 9(3):223–252, 1977.

[24] S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, and T. Anderson. Eraser: A dynamic data race detector for multithreaded programs. *ACM Transactions on Computer Systems*, 15(4):391–411, November 1997.

[25] S.D. Stoller. Testing concurrent Java programs using randomized scheduling. In Klaus Havelund and Grigore Rosu, editors, *Electronic Notes in Theoretical Computer Science*, volume 70. Elsevier Science Publishers, 2002.

[26] C. Szyperski and C. Pfister. Special issues in object-oriented programming - ECOOP 96 workshop reader. In M. Muhlhauser, editor, *Workshop on Component-Oriented Programming, Summary*. dpunkt Verlag, Heidelberg, 1997.

[27] R.N. Taylor, D.L. Levine, and C.D. Kelly. Structural testing of concurrent programs. *IEEE Transactions on Software Engineering*, 18(3):206–215, 1992.

[28] W. Visser, K. Havelund, G. Brat, and S. Park. Model checking programs. In *Proc. 15th International Conf. on Automated Software Engineering*. IEEE Computer Society, 2000.