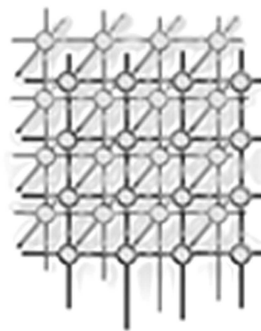

A method for verifying concurrent Java components based on an analysis of concurrency failures



Brad Long^{1,2}, Paul Strooper^{1,*} and Luke Wildman¹

¹ School of Information Technology and Elec. Eng., The University of Queensland, Brisbane, Qld 4072, Australia.

² Australian Development Centre, Oracle Corporation, 300 Ann St, Brisbane, Qld 4000, Australia.

SUMMARY

The Java programming language supports concurrency. Concurrent programs are harder to verify than their sequential counterparts due to their inherent non-determinism and a number of specific concurrency problems, such as interference and deadlock. In previous work, we have developed the ConAn testing tool for the testing of concurrent Java components. ConAn has been found to be effective at testing a large number of components, but there are certain classes of failures that are hard to detect using ConAn. Although a variety of other verification tools and techniques have been proposed for the verification of concurrent software, they each have their strengths and weaknesses.

In this paper, we propose a method for verifying concurrent Java components that includes ConAn and complements it with other static and dynamic verification tools and techniques. The proposal is based on an analysis of common concurrency problems and concurrency failures in Java components. As a starting point for determining the concurrency failures in Java components, a petri-net model of Java concurrency is used. By systematically analysing the model, we come up with a complete classification of concurrency failures. The classification and analysis are then used to determine suitable tools and techniques for detecting each of the failures. Finally, we propose to combine these tools and techniques into a method for verifying concurrent Java components.

KEY WORDS: *Concurrency, verification, testing, component, Java*

Introduction

A concurrent program consists of 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

*Correspondence to: Paul Strooper, School of Information Technology and Elec. Eng., The University of Queensland, Brisbane, Qld 4072, Australia. E-mail: {brad, pstroop, luke}@itee.uq.edu.au
Contract/grant sponsor: ARC Discovery Grant; contract/grant number: DP0343877



cooperate by communicating using shared variables or message passing. Programming and verifying 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. Specific concurrency issues such as interference and deadlock further complicate the verification of concurrent programs, especially since the inherent non-determinism may make it hard to detect these problems through traditional verification techniques.

In this paper, we simplify the problem by focusing on one particular type of concurrent software, namely concurrent Java components. By focusing on verifying an individual component we do not need to be concerned with the number of threads that are executing in the system as a whole, because we assume the component can be accessed by any number of threads at a time. That is, we verify a component under the assumption of multiple thread access. Following Szyperski [42], 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.

In [32], we describe the ConAn (Concurrency Analyser) tool for the testing of concurrent Java components. The approach is based on Brinch Hansen's work [5] on testing Concurrent Pascal monitors. We extended Brinch Hansen's method to deal with Java components and added tool support. Although we found the method and tool support effective in testing a large number of concurrent Java components [32], there are certain classes of failures that are hard to detect using ConAn.

The aim of this paper is to come up with a method for verifying concurrent Java components that includes ConAn and complements it with other static and dynamic verification tools and techniques. One of the main advantages of ConAn is that it extends the tools and techniques for testing sequential Java components and as such it can be used by software developers and testers in industry without the need for special training or advanced theoretical concepts. The method we propose in this paper is similarly practice-oriented.

We derive our method by analysing common concurrency problems and specific concurrency failures in Java components. To analyse the concurrency failures in Java components, we first develop a petri-net model of Java concurrency. Petri-nets [37] are used because they provide a convenient mechanism for modeling the locking of objects. A HAZOP style analysis [6] is applied to determine the possible failures that can be associated with each of the transitions in the petri-net model. We thus use the model to classify the failures that can occur in concurrent Java components and determine suitable tools and techniques for each class of failure. The results of the analysis are then used to combine these strategies into a general method for the verification of concurrent Java components. An obvious area for future work is the empirical evaluation of the method.

Overview of concurrency in Java

The basic Java concurrency constructs are briefly reviewed in this section. Further information regarding concurrency in Java can be obtained from [18, 19, 28, 29].

In the Java programming language mutual exclusion is achieved by a thread locking an object. There are two ways of locking an object: 1) explicitly define a synchronised block and designating the object to lock, or 2) synchronise a method. Synchronising a method is the same as locking the `this` object in a synchronised block. Two threads cannot lock the same object at the same time, thus providing mutual



exclusion. A thread that cannot access a synchronised block because the object is locked by another thread is *blocked*. A thread can lock more than one object by nesting synchronised blocks.

Threads that have a lock on an object can suspend by calling the Java `wait` method on that object. This causes the lock on the object to be released, allowing other threads to obtain the lock. Suspended threads remain dormant until woken. As an example, a particular implementation of the well-known 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.

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 join the set of blocked threads and attempt to regain the object lock for re-entry to the synchronised 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 threads waiting on the object.

Common concurrency problems

We now describe common problems that occur in concurrent programs. This establishes basic terminology for the remainder of the paper.

In concurrent programs, two types of correctness properties are generally distinguished: safety and liveness. Safety properties must always be true. Liveness properties must eventually be true. The most common safety property is mutual exclusion. Violation of this property may result in *interference*. Another important safety property is *absence of deadlock*. Common liveness properties are *process termination* and the requirement that a request by a process to access a shared resource is eventually granted. Common concurrency problems related to these properties are described in more detail below.

Interference occurs when two or more concurrent threads access a shared variable and when at least one access is a write, and the threads use no explicit mechanism to prevent the access from being simultaneous [40]. If a program has potential interference, then the effect of the conflicting access to the shared variable will depend on the interleaving of the threads. Interference is also known as a *data race* or *race condition*.

A system that cannot respond to any signal or request is *deadlocked* [4]. Two forms of deadlock are *deadly embrace* and *dormancy*. Deadly embrace is usually described as a situation where two or more processes are unable to proceed because each is waiting for one of the others to do something in a deadlock cycle [15]. For example, this occurs when a thread holds a lock that another thread desires and vice-versa. Dormancy occurs when a non-runnable thread fails to become runnable [28, page 57]. For example, in Java, this occurs when a `wait` is never balanced by a `notify` or `notifyAll`. This can also occur when a thread is waiting (via `join`) for a non-terminating thread to finish.

Note that if a thread depends on another thread to complete for its own completion, then it will deadlock if the other thread never completes. This may occur if all threads participate in a deadlock cycle. Threads that are waiting on threads in a deadlock cycle are blocked and are called *deadlock-blocked* threads [36].

Livelock is similar to deadlock in that the program does not make progress. However, in a deadlocked computation there is no possible execution sequence which succeeds, whereas in a livelocked



computation there are successful computations, but there are also one or more execution sequences in which no thread makes progress [4]. This situation can arise when two or more processes continuously change their state in response to changes in the other process(es) without doing any useful work [15].

Starvation is related to contention. Contention is when two or more processes compete for the same resource. Absence of starvation is a liveness property that states that a program should eventually gain access to a requested resource. An example of starvation is when a thread tries to access a synchronised block and the JVM always gives the lock to some other waiting thread. The Java specifications [19, 29] do not enforce an order in which threads enter a synchronised block if two or more are trying to do so simultaneously.

Tools and techniques for verifying concurrent software

In this section, we review tools and techniques for the verification of concurrent software, with an emphasis on tools and techniques that can be applied to concurrent Java components.

Code inspections are based on peer reviews by small teams and have been widely used in industry with impressive results [12, 13, 17, 39]. Checklists can be used to enhance the effectiveness of inspection meetings, but we are not aware of any specific checklists for the inspection of concurrent software. However, patterns for bugs in concurrent software [14, 22] could form the basis for such checklists.

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 [26, 33, 35, 43]. The resulting models are then analysed to generate suitable test cases, to generate suitable synchronisation sequences for testing, or to verify properties of the program.

Jlint [2, 27] and FindBugs [22] are static analysis tools that attempt to find bugs in Java programs based on commonly occurring bug patterns. Both are easy to use and apply in practice, but as with other static analysis tools, sometimes problems are reported when none exist and sometimes these tools fail to report a problem when one does exist.

Model checking has been an important research topic in static analysis in recent years. Models of software are often based on finite state machines or call graphs with well-defined mathematical properties. Approaches based on model checking include Bandera [8, 20], JPF (Java Pathfinder) [21, 44], and FSP (Finite State Processes) [34]. These tools are not considered further in this paper and the method, because their use currently still requires knowledge of concepts such as process algebra or temporal logic, which goes against our desire for a practice-oriented method.

Dynamic analysis involves executing a program. Deterministic testing of concurrent programs [3, 5, 7] requires forced execution of the program according to an input test sequence.

In previous work, we developed ConAn to support the testing of Java monitors [32]. ConAn provides tool support for an approach to testing concurrent components that extends Brinch Hansen's approach to testing Concurrent Pascal monitors [5]. ConAn instantiates threads that execute calls to methods of a concurrent component in a sequence defined by the tester. The ConAn generated test driver can be run in a similar manner to test drivers of traditional unit testing and regression testing tools [9, 25].

Most dynamic testing tools in practice are limited to executing many threads against the component under test, whilst attempting to create as many interleavings (of statements between those threads)



as possible, and thereby attempting to force a failure. Some dynamic analysis tools help to increase the chance of failure detection. ConTest [10, 23] and RaceFinder [11] are tools whose main aim is to find data races, deadlocks and other intermittent bugs in multithreaded Java programs. They transform a Java program into a program that should behave in the same way but is more likely to exhibit concurrent bugs such as race conditions and deadlocks. JProbe Threadalyzer [41] provides a graphical means to visually inspect the state of executing threads. It analyses Java code with the aim of pinpointing the cause of stalls, deadlocks and race conditions. Any detected problems are linked back to the location in the source code.

One of the features of Junit++ is that it extends the JUnit testing framework [16] by providing a tester with the ability to execute multiple threads multiple times. Traditional testing tools often include this type of extension to support load testing, which, given sufficient output checking, has been known to detect interference and deadlock. However, these extensions generally do nothing to force specific failures.

A model of Java concurrency

We now propose a general model of the states of a single thread with respect to a single synchronised object. The model is then analysed in the next section for deviations that could cause the common concurrency problems discussed earlier. We are not interested in modelling specific concurrent programs/components. Our model is general enough to capture all multi-threaded programs/components using Java synchronised locks and we analyse the model for what can go wrong for a single point of failure.

Figure 1 models the states of a single thread with respect to a synchronised object by using a petri-net diagram [37]. 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 models the state of the thread with respect to a single object only. If a thread accesses several synchronised objects then it may be in different states with respect to the different objects. Multiple copies of the diagram would be needed to represent this. However, as we only consider single sources of failure, we analyse the behaviour of the thread with respect to one object only.

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.

Places *A* to *D* represent the current state of a thread with respect to an object lock. The marker in place *A* represents a thread that has not yet requested a lock on an object, for example, executing outside a critical section. "Critical section" refers to a code sequence that accesses shared variables. We use the phrase "Outside a critical section" to refer to code that does not access shared variables and therefore does not require exclusive access. This terminology is used rather than "synchronised block" to allow the possibility that a thread may execute inside a critical section without acquiring the appropriate lock. Note that a thread that accesses a shared resource without requesting exclusive access via a lock would remain in place *A*. A marker in place *B* represents a thread requesting a lock on an

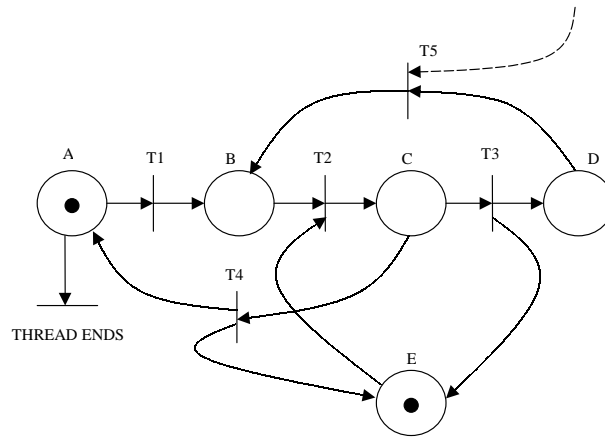


Figure 1. Petri-net model of concurrency

object. A marker in place *C* represents a thread holding a lock. Normally, this would be the case while a thread accesses shared variables in a critical section. A marker in place *D* represents a thread in the *wait* state.

The marker in place *E* represents the availability of the object lock. This place represents a state of the object lock rather than a state of the thread. The object lock is obviously used to control multiple threads each of which is represented by its own diagram but which all share place *E*.

Transition *T1* represents a thread moving from a state of executing outside a critical section to the state of requesting a lock for exclusive access to a critical section. The transition is under the control of the thread itself. Transition *T1* is fired by a thread calling a synchronised method or entering a synchronised block.

Transition *T2* represents a thread moving from a state of requesting a lock to a state of having a lock. Transition *T2* is fired by the JVM serving an object lock to the requesting thread and depends on the availability of the lock. If the thread is requesting a lock (a marker exists at place *B*) and if an object lock is available (a marker exists at place *E*), the thread can obtain the lock (a marker is placed at *C*).

Transition *T3* represents a thread entering the wait state from the locked state. This transition is controlled by the thread and occurs when the `wait` method is called. If the thread holds a lock (marker at *C*), then it may move to the wait state (marker placed at *D*), which also releases the object lock (marker placed at *E*).

Transition *T4* represents a thread leaving the locked state. This transition is controlled by the thread and models what happens when a thread releases the lock on an object. If the thread holds the lock (marker at *C*), then it may exit the critical section (marker placed at *A*) and release the lock (marker placed at *E*).

Transition *T5* represents a thread moving from the waiting state to the state of requesting a lock. This transition is controlled by the JVM in response to another thread calling `notify` or `notifyAll`.



When the thread is waiting (marker at D) and another thread notifies the waiting thread (the incoming dashed arc), then the thread tries to reacquire the object lock it was waiting on (marker placed at B). This has the obvious implication that a thread in the wait state cannot wake itself.

To simplify the model and analysis, some constructs and peculiarities of the Java concurrency model have not been incorporated into the petri-net model. Thread creation, `join`, and `sleep` have not been discussed since these are not typically found in concurrent components themselves, but in the multithreaded programs that use these components, and they do not require any changes to the model presented. The methods `suspend`, `resume` and `stop` are not discussed because they are deprecated and their use is discouraged [24]. Thread interrupts are partially addressed in that interrupt management is the responsibility of the component and it is treated like any other construct in the language. However, to properly deal with interrupts, it should be noted that a thread interrupt can cause a transition from place D to either place B (similar to transition T5) or to place A . Both of these transitions would require a second arc from another thread (again, similar to transition T5), which represents a thread that interrupts the waiting thread. The transition to B would fire when the exception is handled in a try-catch block inside the synchronised section of code. The transition to A would fire when the exception is handled by code outside the synchronised section of code (including the case where the exception is thrown by a synchronised method). Similarly, the model does not currently handle timed waits, but could easily be extended to deal with these. Normally, a thread is only woken up when it is waiting if it is notified (or interrupted) by another thread, which is modelled by transition T5. However, in the most recent Java API documentation, it is acknowledged that there may also be “spurious wakeups”, where a thread wakes up without a corresponding notify or interrupt. Our petri-net model does not incorporate spurious wakeups. Finally, our petri-net model avoids issues related to recursive locks by modelling the requesting, holding and releasing of a lock rather than the entry and exit of synchronised blocks.

Analysis of concurrency failures in Java

We now consider the ways in which concurrency problems may arise from deviations to the thread behaviour captured by the model. Following techniques of hazard/safety analysis, failure conditions are identified for each of the transitions. This approach is taken to ensure that all failures are identified and classified. Using a HAZOP style of analysis [6], we analyze each transition for two deviations using the following “guide phrases”: 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 I summarises the results of applying a HAZOP technique to the analysis of transition failures. Each transition is analysed in turn for the **FF** - *failure-to-fire* and **EF** - *erroneous-firing* deviations. An *interpretation* is given for the deviation in terms of the Java concurrency model. *Failure-to-fire* is interpreted to mean that the transition does not occur at all. *Erroneous firing* is interpreted to mean that the transition occurs when it should not, i.e., it does not occur at the correct time. With *failure-to-fire* we focus on the effect of the transition *not occurring*, whereas with *erroneous firing* we focus on the effect of the transition *occurring*.

The *cause* of the deviation is dependent on whether the transition is under the control of the thread itself (T1, T3, T4) or whether the transition is controlled by the JVM (T2, T5). In the case of the former, the source of the deviation is obviously in the thread code and possible code faults are considered. In the case of the latter, the source of the deviation is purely a *co-effector* and lies either in the JVM or



Deviation	Interpretation	Cause	Co-effectors	Consequences	Verification Notes
FF-T1	Thread never requests a lock.	1. No use of <code>synchronize</code> . 2. Locks wrong object.	Another thread accesses the shared variable simultaneously.	Interference.	Code inspection, static analysis
EF-T1	1. Requests access too early. 2. Requests access too late.	1. <code>synchronize</code> used before access required. 2. Earlier statements access shared variables.	1. None. 2. See FF-T1.	1. Delay. 2. See FF-T1.	Code inspection, static analysis (FindBugs)
FF-T2	Thread never receives a lock.	1. Lock held by another thread. 2. JVM never serves lock to thread.	1. Thread owning lock never releases it. 2. JVM unfair.	1. Deadlock. 2. Starvation.	Code inspection, static analysis (Jlint)
EF-T2	Lock acquired too early or too late.	JVM failure.	Not applicable.	Not applicable.	Not applicable.
FF-T3	Thread never suspends.	Call to <code>wait</code> does not occur.	None.	Incoincidence.	Code inspection, Dynamic analysis (ConAn)
EF-T3	1. Thread suspends too early. 2. Thread suspends too late.	Incorrect time to call <code>wait</code> .	None.	Incoincidence.	Dynamic analysis (ConAn)
FF-T4	Thread never leaves critical section (CS).	1. Endless loop or awaiting input. 2. Waits instead. 3. Loops on wait. 4. Requests lock (see FF-T2).	1. None. 2. None. 3. Competing threads always get priority. 4. See FF-T2.	1. Co-effector for FF-T2. 2. Incoincidence. 3. Starvation. 4. See FF-T2.	Dynamic analysis (ConAn)
EF-T4	1. Leaves CS too early. 2. Leaves CS too late. 3. Leaves CS instead of waiting.	1. Subsequent statements access shared variables. 2. No shared access required on prior statements. 3. See FF-T3.	1. Another thread accesses shared variable simultaneously. 2. None. 3. See FF-T3.	1. Interference. 2. Delay. 3. See FF-T3.	Code inspection, Static Analysis, Dynamic analysis (ConAn)
FF-T5	Thread is never notified.	1. No thread calls <code>notify</code> . 2. <code>notify</code> vs. <code>notifyAll</code> . 3. Unfair selection by JVM.	1. None. 2. Progress depends on notified thread. 3. Always another thread waiting.	1. Dormancy. 2. Dormancy. 3. Starvation. In all cases, co-effector for FF-T2 if lock held.	Dynamic analysis (ConAn)
EF-T5	Thread is notified too early or too late.	Notification occurs at wrong time.	None.	Incoincidence.	Dynamic analysis (ConAn)

Table I. Concurrency failure classification



another thread. In the case of the JVM, we assume that the JVM conforms to its specification but we do not assume it is fair.

The *co-effectors* describe any extra conditions that may be required for one of the hazardous *consequences* to arise. When dynamically testing for the occurrence of the hazard, the co-effectors will have to be produced also. We consider the following hazardous *consequences*.

- Interference.
- Deadlock either through deadly embrace or deadlock-blocking.
- Starvation through never receiving the lock.
- Livelock through continued execution without releasing the lock.
- Dormancy through waiting without notification.
- Incoincidence (from ‘incoincident’ – not agreeing in time, in place, or principle [38]) through a call completing at the wrong time (excluding consequences already listed above).

In addition, efficiency concerns such as delays through unnecessary synchronisation are noted. Where particular causes, co-effectors, or consequences are related, numeric labels are used to link them.

Under *Verification Notes*, practical tools and techniques for detecting the failure are listed. This takes the place of *mitigations* in a traditional HAZOP table. Such tools and techniques will be employed in the combined method described in the following section.

A detailed explanation for each entry in the table can be found in [30]. Space limitations prevent us from giving a complete description here. However, as an example we now explain the analysis of the failure-to-fire deviation of transition T1.

Failing to fire transition T1 means the thread fails to enter a synchronised block for mutually exclusive access to the shared resource. Failure to gain mutually exclusive access can occur in two ways: 1) the critical section is not in a synchronised block, or 2) the wrong object is locked. This could potentially lead to interference if more than one thread accesses the shared resource simultaneously. It is possible to check for these conditions using either code inspection, static analysis, or dynamic analysis, but each technique has its problems.

Visual code inspection for interference is feasible for simple components, but becomes difficult for complex components that make call-outs to other classes. Static analysis is the most effective way to check for interference. Tools such as Jlint [2, 27], Findbugs [22], and JPF [21, 44] would be applicable. Unfortunately, references to objects cannot always be evaluated statically. This makes static analysis very difficult for complex programs. Dynamic analysis cannot guarantee that interference will be detected because it cannot guarantee that all interleavings of threads will be executed. Because code inspection and static analysis are straightforward to apply and cover most of the cases, these have been included in the method.

Having used the HAZOP analysis to develop a comprehensive list of possible faults and a corresponding list of tools and techniques for verifying the absence of those faults, we now present a combined method for verifying concurrent Java components that addresses all faults.

A method for verifying concurrent Java components

We now propose a 9-step method for verifying multi-threaded components that can be used by software developers and testers in industry without the need for special training or advanced theoretical concepts.



The method is generic and designed without consideration of the context in which the component is or will be used. For instance, if the component were to be used in a safety-critical context then a more comprehensive verification strategy would be required.

We structure the method by the hazardous consequences rather than the transition deviations of the previous section because the consequences of some of the transitions overlap. This also produces a method that is expressed independently of the petri-net model.

The first 4 steps of the method are concerned with detecting interference. The method detects interference using a combination of code inspection and automatic static analysis. This addresses deviations FF-T1, EF-T1(2), and EF-T4(1) from the table. The next 2 steps are concerned with detecting a “deadly embrace” style deadlock and a combination of code inspection and automatic static analysis is used. This addresses deviations FF-T2(1) and FF-T4(1,4). The final 3 steps of the method are concerned with testing that the component under analysis meets its functional requirements, and a combination of code inspection and automatic static and dynamic analysis is used to achieve this. This addresses deviations EF-T1(1), FF-T2(2), FF-T3, EF-T3, FF-T4(2,3), EF-T4(2,3), FF-T5(1,2,3), and EF-T5.

Step 1. Execute FindBugs and review any warnings of “inconsistent synchronization”. FindBugs searches Java classes for instances of so-called *bug patterns*. A bug pattern is a code segment that may be erroneous. The tool reports a list of the classes analysed and the bug patterns detected, reporting their location in the source code. To assist in the detection of interference, the tool should be used with the *LockedFields* detector on, and then code inspection should be used (see Step 2) to determine if the bug pattern is, in fact, an error. The inspection step is important because the tool may, in certain circumstances, fail to identify a synchronisation fault, since not all faults can be detected by automatic static analysis.

Step 2. Ensure that shared variables are protected by synchronised blocks. This step uses code inspection to check for interference caused by unsynchronised access to shared variables. The code is inspected to ensure that a shared variable is always accessed within a synchronised block, and that each such access to that variable is synchronised on the same object. Any `static` shared variable must be synchronised on a `static` (or class) lock.

Step 3. Ensure that lock references are not reassigned. A synchronised block of code names the object that it locks during the execution of the block. The object is named in the usual way via a reference variable. In this step, code inspection is used to check that the lock reference is not reassigned. Effectively, we want the lock reference to be constant and always refer to the same lock object. To this end, all lock variables should be declared `final`. Note that this step is trivial if all synchronised blocks are implemented as synchronised methods.

Step 4. Ensure that shared variables are properly encapsulated. An instance variable that has public or package scope can be accessed by any thread that accesses its object. If the variable is shared then this presents an obvious loophole to Step 2. In this step, code inspection is used to ensure that shared variables are properly encapsulated.

Step 5. Execute Jlint and review warnings of deadlock cycles. Jlint can find errors in Java classes by static analysis of byte code. In the case of a complex component it is very useful for automatically calculating lock graphs. However, if this analysis results in the reporting of deadlock cycles, code



inspection as detailed in Step 6 must still be used to confirm this finding, because in some circumstances Jlint reports faults where none exist.

Step 6. Build a lock graph to check the absence of deadlock cycles. To build a lock graph, first a node is created for each distinct lock that is used to synchronise a block (note that synchronised methods lock `this`). Then for each pair of nested synchronised blocks, a directed edge is drawn from the node that represents the outer lock to the node that represents the inner lock. The existence of a cycle in this graph confirms the possibility of a deadly embrace.

Step 7. Check the use of notifications. A call to `notify` awakes at most one thread, however, fairness considerations typically require the notification of all waiting threads using `notifyAll`. This step involves code inspection to check the appropriate use of `notify` or `notifyAll`. FindBugs can be used to perform this check, but code inspection is still required to determine if the “bug pattern” really does represent a fault.

Step 8. Check the entry and exit conditions of condition synchronisations. A condition synchronisation is typically implemented as a ‘wait loop’ within a synchronised block (`while (C) {wait();}`). The loop has an entry condition C and an exit condition $\neg C$. The loop exit condition $\neg C$ represents the desired synchronisation condition, that is, the purpose of the wait loop is to delay thread execution until $\neg C$ is true. The FindBugs tool is able to detect ‘bug patterns’ that involve calls to method `wait`, but, once again, code review is required to determine the presence of an actual fault.

Step 9. Use ConAn to test the functional behaviour of the component and the call completion times. ConAn provides tool support for an approach to testing concurrent Java components that consists of three steps. The first step is to *identify test conditions* that will exercise each monitor operation under test. Test conditions should be included to ensure loop coverage of the code under test, consideration for the number and type of processes suspended inside the monitor, and significant state and parameter values. The second step is to *construct test sequences* of operation calls that will exercise each operation for each of the test conditions identified in step one. The test sequences are specified as ConAn test sequences, from which ConAn automatically generates a test driver that uses an abstract clock to impose the test sequence by controlling the execution of threads. The third step is to *execute* the test driver generated from the test sequences. The tool also compares actual outputs against expected outputs as specified in the test sequences, including completion times of calls (to detect incoincidence).

Conclusion

The non-deterministic nature of concurrent programs means that conventional verification tools and techniques are inadequate. New techniques and tools need to be developed to allow the verification of such programs. In this paper, we extend an approach to testing concurrent Java components to a more comprehensive method for verifying such components, based on an analysis of a petri-net model of Java concurrency. 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 then used to determine appropriate verification tools and techniques for each of the concurrency failures, which are then combined in the proposed method.



The obvious area for future work is to apply the method on a number of concurrent components to evaluate its effectiveness. An initial exploration of the method on an implementation of the readers-writers problem and several mutants of that implementation is presented in [31]. Undoubtedly, as new tools and techniques are developed and mature, the method will need to be updated, but the framework we have presented in this paper can then still be used to determine if and how such a tool would best fit in the overall method. Another area for future work is to extend the petri-net model and the analysis to eliminate the limitations of the model discussed earlier. We have already extended the ConAn test case selection strategy and tool support to deal with interrupts and timed waits [45].

ACKNOWLEDGEMENTS

This article has benefited from proof-reading by Roger Duke and Doug Goldson, and discussions with Neil Robinson and Andrew Rae.

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 Conference*, pages 68–75. IEEE Computer 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. M. Ben-Ari. *Principles of Concurrent and Distributed Programming*. Prentice Hall, 1990.
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. R.H. Carver and K-C. Tai. Use of sequencing constraints for specification-based testing of concurrent programs. *IEEE Transactions on Software Engineering*, 24(6):471–490, 1998.
8. C. Corbett, M. Dwyer, J. Hatcliff, S. Laubach, C. Pasareanu, Robby, and H. Zheng. Bandera: extracting finite-state models from Java source code. In *Proceedings 22nd International Conference on Software Engineering*, pages 439–448. IEEE Computer Society, 2000.
9. N. Daley, D.M. Hoffman, and P.A. Strooper. A framework for table driven testing of Java classes. *Software – Practice and Experience*, 32:465–493, 2002.
10. O. Edelstein, E. Farchi, Y. Nir, G. Ratsaby, and S. Ur. Multithreaded Java program test generation. *IBM Systems Journal*, 41(1):111–125, 2002.
11. Y. Eytani, E. Farchi, and Y. Ben-Asher. Heuristics for finding concurrent bugs. In *Proceedings of the 17th International Parallel and Distributed Processing Symposium (IPDPS 2003) – 1st International Workshop on Parallel and Distributed Systems: Testing and Debugging*. IEEE Computer Society, 2003.
12. M.E. Fagan. Design and code inspections to reduce errors in program development. *IBM System Journal*, 15(3):182–211, 1976.
13. M.E. Fagan. Advances in software inspections. *IEEE Transactions on Software Engineering*, 12(7):744–751, July 1986.
14. E. Farchi, Y. Nir, and S. Ur. Concurrent bug patterns and how to test them. In *Proceedings of the 17th International Parallel and Distributed Processing Symposium (IPDPS 2003) – 1st International Workshop on Parallel and Distributed Systems: Testing and Debugging*. IEEE Computer Society, 2003.
15. Free online dictionary of computing. Supported by the Department of Computing, Imperial College of London. Available online at <http://foldoc.doc.ic.ac.uk/foldoc/index.html>, 2003.
16. E. Gamma and K. Beck. JUnit testing framework. Available online at <http://www.junit.org/>, 2002.
17. T. Gilb and D. Graham. *Software Inspection*. Addison-Wesley, 1993.
18. J. Gosling and K. Arnold. *The Java Programming Language*. Addison Wesley, 2nd edition, 1998.
19. 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>.



20. J. Hatcliff and M. Dwyer. Using the Bandera tool set to model-check properties of concurrent Java software. In *Proceedings 12th International Conference on Concurrency Theory (CONCUR'01)*, pages 39–58. Springer-Verlag, 2001.
21. K. Havelund. Java PathFinder, a translator from Java to Promela. In *Proceedings of 5th and 6th SPIN Workshops*. Springer-Verlag, 1999.
22. D. Hovemeyer and W. Pugh. Finding bugs is easy. <http://www.cs.umd.edu/~pugh/java/bugs>.
23. IBM. Contest. Available online at <http://www.haifa.il.ibm.com/projects/verification/contest/index.html>, 2003.
24. 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>, 2002.
25. R. Jeffries. Extreme testing. *Software Testing and Quality Engineering*, pages 23–26, March 1999.
26. 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.
27. K. Knizhnik and C. Artho. Jlint manual. Available online at <http://artho.com/jlint>, 2002.
28. D. Lea. *Concurrent Programming in Java*. Addison Wesley, 1997.
29. T. Lindholm and F. Yellin. *The Java Virtual Machine Specification*. Addison Wesley, 2nd edition, 1999. Also online at <http://java.sun.com/docs/books/vmspec/index.html>.
30. B. Long. *Testing Concurrent Java Components*. PhD thesis, University of Queensland, 2005.
31. B. Long, R. Duke, D. Goldson, P. Strooper, and L. Wildman. Mutation-based exploration of a method for verifying concurrent java components. In *Proceedings of the 18th International Parallel and Distributed Processing Symposium (IPDPS 2004) – 2nd International Workshop on Parallel and Distributed Systems: Testing and Debugging*. IEEE Computer Society, 2004.
32. B. Long, D. Hoffman, and P. Strooper. Tool support for testing concurrent Java components. *IEEE Transactions on Software Engineering*, 29(6):555–566, June 2003.
33. 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.
34. J. Magee and J. Kramer. *Concurrency: State Models and Java Programs*. John Wiley & Sons, 1999.
35. 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.
36. Y. Nonaka, K. Ushijima, H. Serizawa, S. Murata, and J. Cheng. A run-time deadlock detector for concurrent Java programs. In *Proceedings of the 2001 Asia-Pacific Software Engineering Conference*, pages 45–52. IEEE Computer Society, 2001.
37. J.L. Peterson. Petri nets. *ACM Computing Surveys*, 9(3):223–252, 1977.
38. N. Porter, editor. *Webster's Revised Unabridged Dictionary*. C. & G. Merriam Co., 1913.
39. G.W. Russell. Experience with inspection in ultralarge-scale developments. *IEEE Software*, pages 25–31, January 1991.
40. 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.
41. Quest Software. JProbe Threadalyzer. Available online at <http://java.quest.com/jprobe/threadalyzer.shtml>, 2003.
42. C. Szyperski. *Component Software: Beyond Object-Oriented Programming*. Addison Wesley, 1998.
43. 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.
44. W. Visser, K. Havelund, G. Brat, and S. Park. Model checking programs. In *Proceedings of the 15th International Conference on Automated Software Engineering*, pages 3–12. IEEE Computer Society, 2000.
45. L. Wildman, B. Long, and P. Strooper. Testing Java interrupts and timed waits. In *Proceedings of the 11th Asia-Pacific Software Engineering Conference*, pages 438–447. IEEE Computer Society, 2004.