# A State Exploration-Based Approach to Testing Java Monitors

Yu Lei[1], Richard Carver[2], David Kung[1], Vidur Gupta[1], Monica Hernandez[1]

[1]Dept. of Comp. Sci. and Engineering
University of Texas at Arlington
Arlington, TX 76019-0015
{ylei, kung, gupta, hernandez}@cse.uta.edu

[2]Dept. of Computer Science
George Mason University
Fairfax, VA 22030
rcarver@cs.gmu.edu

## Abstract

A Java monitor is a Java class that defines one or more synchronized methods. Unlike a regular object, a Java monitor object is intended to be accessed by multiple threads simultaneously. Thus, testing a Java monitor can be significantly different from testing a regular class. In this paper, we propose a state exploration-based approach to testing a Java monitor. A novel aspect of our approach is that during exploration, threads are introduced on-the-fly, and as needed, to simulate race conditions that can occur when multiple threads try to access a monitor object at the same time. Furthermore, each transition is defined in a way such that the behavior of the threads along each path can be precisely characterized and controlled. We describe a prototype tool called *MonitorExplorer* and report three case studies that are designed to provide an initial evaluation of our approach.

## 1. Introduction

Multithreaded programming has become commonplace in modern software development. Using multiple threads increases the responsiveness of user interfaces. While one thread is performing computational tasks, another thread can respond to user inputs. More importantly, many problems can be solved more naturally and efficiently by creating multiple threads. As an example, a web server typically creates separate threads to service incoming client requests.

An important feature of the Java language is that it provides built-in support for multithreaded programming. The Java core library includes a class named *Thread* as a programming abstraction of a thread. The *Thread* class defines a set of operations that are commonly performed on a thread. For thread synchronization, Java provides a simplified implementation of the monitor construct, which we refer to as a Java monitor [11]. Syntactically, a Java monitor is a Java class that defines one or more synchronized methods, i.e., methods whose signatures contain the keyword synchronized. In general, there are two types of thread synchronization: *mutual exclusion* and *condition synchronization*. Mutual exclusion ensures that at any given time, at most one thread can execute inside a critical section. (Recall that a critical section is a fragment of code that accesses shared data.) Condition synchronization ensures that a thread can proceed if and only if a certain condition is satisfied, e.g., a buffer is not full, a resource is not in use, etc. The Java runtime environment automatically enforces mutual exclusion on the synchronized methods in a Java monitor. Condition synchronization can be programmed in a Java monitor using wait and notify statements, which allow threads to be blocked and awakened inside a synchronized method.

Over the years, many approaches have been developed for testing regular classes [4][9]. In these approaches, a test case is a sequence of method calls that are issued by a single-threaded test driver. These approaches cannot be directly applied to test a Java monitor. This is because unlike the methods of a regular object, which are supposed to be called by at most one thread at a time, the methods of a Java monitor object are intended to be called by multiple threads simultaneously. Thus, in order to simulate the possible scenarios in which a Java monitor object may be used, it is necessary to create more than one thread in each test case. Furthermore, if a single driver thread were to be used to execute a test case, the driver thread could be blocked by a synchronized method and could thus be prevented from completing the sequence of method calls in the test case. The need for creating multiple threads in a test case, however, raises the following two issues:

- How many threads should be created in each test case? Many synchronization faults can only be detected when a certain minimum number of threads interact. However, the necessary number of threads is often not known *a priori*.
- Since multiple threads are used, each test run may exhibit non-deterministic behavior. How should the desired behavior of each test run be specified? And given a desired behavior, how can each test run be controlled so that the desired behavior is exercised?

The main contribution of this paper is a state exploration-based approach that addresses the above issues. This approach involves systematically exploring the state space of a Java monitor. Each explored path (from the initial state

to a state where the exploration backtracks) can be considered as a dynamically constructed test case. A novel aspect of our approach is that threads are introduced on-the-fly, and as needed, along each path during exploration. The rules for deciding when to introduce a new thread are defined to simulate the race conditions that can occur when multiple threads try to access a monitor object simultaneously. A state transition is defined to occur when a synchronization operation is executed that causes a thread to enter or exit a monitor. Synchronization operations are of a smaller granularity than method calls and allow thread behavior to be precisely characterized and controlled. We describe a prototype tool called *MonitorExplorer*, and report three case studies that provide an initial evaluation of our approach. The results indicate that our approach is very effective in detecting synchronization-related faults for the monitors we have studied.

The rest of this paper is organized as follows. Section 2 surveys related work. Section 3 describes the semantics of Java monitor. Section 4 presents our state exploration-based approach. Section 5 illustrates our approach using an example scenario. Section 6 describes the *MonitorExplorer* tool and discusses some implementation issues. Section 6 also presents the three case studies. Section 7 provides concluding remarks and describes our plan for future work.

## 2. Related Work

As we mentioned in Section 1, the problems of determining the number of threads in a test case and dealing with non-deterministic behavior are problems that do not exist when testing a regular class. Thus, we will only review existing work on testing monitors and on state exploration techniques.

Hansen developed an approach to testing Concurrent Pascal monitors [2]. His approach has three major steps. The first step identifies for each monitor method a set of preconditions that if satisfied, will cause every branch of the method to be executed at least once. The second step constructs a single sequence of monitor calls such that each identified precondition is satisfied at least once. The last step creates a multithreaded test driver to execute the monitor call sequence identified in the second step. Each thread in the test driver executes one or more monitor calls in the sequence. During testing, all the threads are synchronized to ensure that they execute their calls in a specified order.

Hansen's approach was extended in [10] for testing Java monitors. Observing that a wait statement in a Java monitor often needs to be put inside a loop (i.e., instead of an if statement), the authors extended Hansen's approach to achieve loop coverage, in addition to branch coverage. That is, the identified preconditions are required not only to cause every branch to be executed but also to cause every loop to be executed zero time, one time, and more than one time. Carver and Tai [3][5] generalized Hansen's technique for synchronizing threads during test execution (i.e., the last step in Hansen's technique) and showed how to apply their technique to monitors, semaphores, locks, and message passing.

To the best of our knowledge, Hansen's approach and its extensions are the only existing approaches to testing a monitor. The tool support described in [10] only automates the last step; the first two steps still need to be performed manually. As a result, the above approach can be time consuming and error-prone. In contrast, our approach is more systematic, and the state exploration is conducted in an automatic manner. Furthermore, our approach explores the state space of a Java monitor until a fixed point is reached, which is a different stopping criterion than those provided by branch- and loop-based coverage.

Several state exploration-based approaches have been developed for testing concurrent programs These approaches either directly explore the state space of a concurrent program [7][8], or extract an abstract model from the program and then explore the state space of the model using a model checker such as Spin [6]. An implicit assumption held by these approaches is that they will be applied to a standalone program. Before these approaches can be used to test a *component* like a Java monitor, a test program must be constructed to simulate the possible scenarios in which the Java monitor may be used. However, since failures are often triggered by unexpected scenarios, constructing a test program that will expose the potential faults is a difficult task. This is in contrast to our approach, which introduces test threads as needed to create race conditions that can trigger failures, removing the need to construct a test program.

We wish to point out that all the existing state exploration approaches assume a closed system into which no new threads can be introduced during state exploration. (Some approaches allow the system to create threads dynamically during exploration. However, such a system is still closed in the sense that the types of threads, the number of threads, and the time at which the threads are created are prescribed by the system description.) This is different from our state exploration approach, which treats a Java monitor as a member of an open system where threads are introduced on-the-fly and as needed (i.e., not prescribed statically) during state exploration.

## 3. Java Monitor Semantics

A Java monitor is a Java class that defines one or more synchronized methods. The data members of a Java monitor represent shared data. Threads access the shared data by calling the synchronized methods defined in the monitor. Fig. 1 shows a graphical view of a Java monitor.

It consists of three components: the *entry* queue, the *critical section* (or CS), and the *condition* queue. A synchronized method can only be executed inside the CS. Mutual exclusion to the CS is automatically provided by the Java runtime environment. That is, at any given time, at most one thread is allowed to execute inside the CS. If a thread calls a synchronized method while another thread is executing inside the monitor, the calling thread must wait on the *entry* queue of the monitor.

Condition synchronization is achieved by using the condition queue and operations wait and notify/notifyAll. Only a thread that is already inside the CS can execute wait and/or notify/notifyAll. When a thread executes wait, it releases mutual exclusion and blocks itself on the condition queue, which allows another thread to enter the CS. When a thread executes notify (or notifyAll), it awakens one (or all) of the threads blocked in the condition queue, if the queue is not empty, and then continues to execute inside the CS. An awakened thread does not immediately re-enter the CS. Instead, it joins the entry queue and thus competes with other threads trying to enter/re-enter the CS. Note that according to the Java specification, notify does not necessarily preserve *First-Come-First-Serve* semantics, i.e., it may not awaken the longest waiting thread.

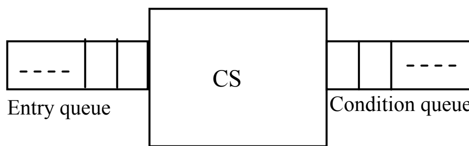


**Figure 1. A graphical view of Java monitor**.

Fig. 2 shows a Java monitor that solves the bounded buffer problem [5]. A producer thread calls *deposit()* to put an integer into the buffer, and a consumer thread calls *withdraw()* to get an integer from the buffer. The correctness requirement dictates that a producer or consumer thread should be blocked when the buffer is full or empty, respectively.

## 4. The State Exploration-Based Approach

Fig. 3 shows an algorithm called *MonitorTest* that implements our state exploration-based approach. Algorithm *MonitorTest* takes as input a Java monitor *M* and an initial state *s0*. It begins by creating an instance *m* of *M* and initializing *m* to state *s0*. Then, it initializes two data structures, namely, *stack* and *visited*. The *stack* stores the transition sequence from the initial state to the state currently being explored. States that have already been explored are added to *visited* so that they are explored only once. A call to function *getAbstractState()* returns an abstraction of the current state, and a call to function *getEnabledTransitions()* returns the set of transitions that are enabled at the current state. As explained later, the set of transitions returned by *getEnabledTransitions()* may also include transitions that introduce new threads. That is, the logic for introducing threads on-the-fly is implicitly encoded in this function. The implementation details of functions *getAbstractState() and getEnabledTransitions()* are discussed in Section 4.1 and 4.2.

We will not explain algorithm *MonitorTest* line by line, as for the most part it is a classical depth-first search algorithm. Instead, we will only make three observations. First, algorithm *MonitorTest* uses abstract states to determine whether a (concrete) state needs to be expanded. As shown in Section 4.1, the abstract state space of a Java monitor is bounded, which ensures that the algorithm will terminate. Second, undoing a transition (line 16) restores the previous state from which the transition was executed. This can be done by re-executing all but the last transition in the transition sequence on the *stack* [7]. Doing so allows us to avoid representing, saving, and restoring explicit representations of concrete states, which, as explained in Section 4.1, can be difficult. Finally, algorithm *MonitorTest* takes as input an initial state, which can be any state of the monitor. In order to ensure adequate test coverage, some systematic strategy such as boundary testing can be used to identify a set of initial states to be

```
class BoundedBuffer {
     private int fullSlots=0;
     private int capacity = 0;
     private int[] buffer = null;
     private int in = 0, out = 0;
     public BoundedBuffer(int bufferCapacity) {
1.       capacity = bufferCapacity;
2.       buffer = new int[capacity];
     }
     public synchronized void deposit (int value) {
3.       while (fullSlots == capacity) {
4.           try { wait(); } catch (InterruptedException ex) {}
         }
5.       buffer[in] = value;
6.       in = (in + 1) % capacity;
7.       if (fullSlots++ == 0) {
8.           notifyAll();
         }
     }
     public synchronized int withdraw () {
9.       int value = 0;
10.      while (fullSlots == 0) {
11.          try { wait(); } catch (InterruptedException ex) {}
         }
12.      value = buffer[out];
13.      out = (out + 1) % capacity;
14.      if (fullSlots-- == capacity) {
15.          notifyAll();
         }
16.      return value;
     }
}
```

**Figure 2. Monitor *BoundedBuffer***

```
Initialize:
1. let stack be an empty stack;
2. let visited be an empty set;
3. create a instance m of M, and initialize m to state s0;

   MonitorTest () {
4.    AbstractState state = getAbstractState ();
5.    add state into visited;
6.    transitions = getEnabledTransitions ();
7.    Explore (transitions);
   }
   Explore (transitions: a set of transitions) {
8.    for (each transition t in transitions) {
9.       push t onto stack;
10.      execute t;
11.      state = getAbstractState ();
12.      if (state is not in visited) {
13.         add state into visited;
14.         Explore (getEnabledTransitions ());
         }
15.      pop t out of stack;
16.      undo t;
      }
   }
```

**Figure 3. Algorithm *MonitorTest***

used.

## 4.1 Function *getAbstractState*

Function *getAbstractState* returns an abstraction of the current state of the state exploration. We will refer to this abstraction as the abstract state of the current state. In the following, we discuss what components should be included in the state representation of a Java monitor, explain the need for state abstractions, and present some guidelines about how to make appropriate state abstractions.

The state of a Java monitor must include all of the information that may affect the future behavior of the monitor (or more precisely, the behavior of the threads that access the monitor). Therefore, the state representation of a Java monitor should include the following components: (1) the values of all the data members; (2) the state of the thread currently inside the CS; (3) the states of the threads in the entry queue; and (4) the states of the threads in the condition queue. Note that (2), (3), and (4) are internal states that are not directly visible to the programmer but that may still affect the behavior of the monitor.

Since there is no bound on the number of threads that may access a Java monitor, the state space based on the above representation is infinite. Therefore, appropriate state abstractions are necessary to ensure that the exploration of the state space will terminate. Moreover, the concrete state representation of a thread can be complicated as the state of a thread needs to include everything that may affect the future behavior of a thread, e.g., the thread's call stack [7]. Appropriate state abstractions are thus also needed to avoid the use of concrete state representations.

In the following, we present some guidelines for making appropriate state abstractions. The abstraction of data members can be made using existing data abstraction techniques [1]. Since we are mainly interested in synchronization faults, we only consider the data members that could affect the synchronization behavior of a monitor. A key observation is that a data member affects the synchronization behavior of a monitor if the data member is directly or indirectly referenced in a branching statement that contains paths that may display different synchronization behavior. Therefore, the abstract values of a data member can be identified by partitioning the domain of the data member into intervals that lead to those different paths. For example, in Fig. 2, the only data member that affects the synchronization behavior of monitor BoundedBuffer is *fullSlots*. The abstract values of *fullSlots* are *0*, *(0, capacity)*, and *capacity*, where *(0, capacity)* indicates an open interval, i.e., $0 < fullSlots < capacity$.

The other components of the state representation are concerned with the threads inside a Java monitor. A key requirement for the abstractions of these components is that the resulting abstract state must be independent from the identities of the threads. Otherwise, the abstract state space will be infinite, as there can be an arbitrary number of threads accessing a Java monitor. While identities must be abstracted away, the abstractions must retain enough information to allow adequate test coverage to be achieved. For this purpose, we introduce the notion of *thread type* to abstract away thread identities. For example, the type of a thread *T* can be characterized by the method that *T* executes. Doing so will identify two types of threads in Fig. 2: (a) *D* – a depositing (or producer) thread that executes *deposit()*; (b) *W* – a withdrawing ( or consumer) thread that executes *withdraw()*. As another example, the type of a thread *T* may also include a flag that indicates whether *T* is a new thread entering the *CS* for the first time or an old thread trying to reenter the *CS* after being notified.

Below we present possible abstractions of the *CS*, entry queue, and condition queue, based on the notion of thread type. These abstractions will be used in our case studies:

- *CS:* The *abstract* state of the CS is empty if no thread is executing inside the CS; otherwise, it is identified by the type of the thread that is executing inside the *CS*.
- *Entry queue:* The *abstract* state of the entry queue is empty if no thread is in the entry queue; otherwise, it is identified by the type of the thread that is at the *front* of the entry queue. If we consider all of the

threads in the entry queue to be competing to enter the monitor, this abstraction captures the result of this competition.

- *Condition queue:* The *abstract* state of the condition queue is empty if no thread is in the condition queue; otherwise, it is identified by a so-called type vector, which contains the different thread types that currently exist in the condition queue. For example, if the condition queue in Fig. 2 contains two threads, with one executing *deposit()* and the other executing *withdraw()*, then its abstract state is *(D, W)*. The order of the elements in a type vector is not significant. The reason is that *notify()* awakens an *arbitrary* thread, and thus the result of executing two *notify()* operations is independent from the order of the threads in the condition queue.

Note that the above abstractions for the CS, the entry queue, and the condition queue can be implemented in an application-independent manner, and thus do not have to be provided by the user. Also note that abstractions can be made at different levels of details. For instance, more information can be encoded in the abstract state of the condition queue so that we can determine whether there are zero, one, or more threads of a certain type in the queue. Typically, the more information contained in the abstractions, the more powerful and the more expensive they are when it comes to fault detection.

## 4.2 Function *getEnabledTransitions*

Fig. 4 shows function *getEnabledTransitions*, which returns the set of transitions that can be executed in the current state of the state exploration. As mentioned earlier, this function is also responsible for introducing new threads. As a result, the set of transitions returned by this function may include transitions that introduce new threads.

Before we explain the details of this function, we introduce our execution model. We consider each thread to execute a sequence of operations, as specified by the text of the monitor method(s) being invoked. An operation is visible if its execution changes the internal state of a Java monitor, i.e., the state of the *CS*, the entry queue, and/or the condition queue. The following visible operations can be performed by a thread:

- *enter*: A thread enters/re-enters the CS.
- *wait*: A thread executes a wait operation.
- *awaken*: A thread awakens one or more threads in the condition queue.
- *exit:* A thread exits the CS.

We consider other operations to be invisible operations. In our approach, each test thread executes one or more synchronized methods. Thus, each thread must start with an *enter* operation and end with an *exit* operation. We define a transition as a visible operation followed by a finite number of invisible operations (until the next visible operation). The type of a transition is determined by the type of the visible operation of the transition.

We point out that the transitions defined above capture every change to the internal state of a monitor. It was shown in [5] that a monitor-based execution can be reproduced by replaying the order in which these transitions are executed. We stress that these transitions are of a finer granularity than method calls. The reason why we cannot define a transition as a method invocation is that a thread may be blocked in the middle of a synchronized method and may later resume execution after being notified. Therefore, method calls cannot be used to precisely characterize the behavior of the threads that access a Java monitor.

One of the novel aspects of our approach is that during state exploration, we allow new threads to be introduced on-the-fly, and as needed, to simulate race conditions that can occur when multiple threads access a Java monitor at the same time. For this purpose, we introduce an auxiliary type of transition, called *introduce*. An *introduce* transition takes a synchronized method as a parameter. Conceptually, the execution of an *introduce* transition will create a new thread that executes the given synchronized method and then put it into the entry queue. In order to minimize the number of threads that have to be created, an *introduce* transition will reuse a thread if that thread was introduced earlier to execute a method and the execution of that method has finished. Note that an *introduce* transition is a visible transition as it changes the internal state of a Java monitor. Also note that unlike other visible transitions, which are executed by a test thread, an *introduce* transition is executed by our state exploration engine.

Now we are ready to explain function *getEnabledTransitions* in Fig. 4. Lines 2 to 4 introduce new threads if necessary. The following two rules determine when new threads are introduced:

a. *If the entry queue and the CS are both empty at the current state, then for each synchronized method, we introduce a new thread to execute the method.*

   This rule ensures that the state exploration will continue when all the existing threads are finished or blocked. Since the introduced threads compete to enter the *CS*, the state exploration engine will explore the possibility that for each synchronized method, a new thread that executes the method wins the competition. Note that this rule can always be applied to an initial state, where

no thread has been introduced yet. Also note that the *CS* becomes empty when the thread inside the *CS* exits, which can be due to the execution of a *wait* operation or to reaching the end of a synchronized method. Also note that the introduction of new threads will eventually come to an end as the current state is not expanded if its abstract state was already visited, and the abstract state space is bounded.

b. *If the entry queue at the current state is empty, and the next operation to be executed by the thread inside the CS is a notify/notifyAll operation, then for each synchronized method, we introduce a new thread to execute the method before the notify/notifyAll operation is executed.*

The motivation for the above rule is as follows. When a notify operation is executed, an awakened thread *T* in the condition queue will join the entry queue and will compete with other threads to reenter the monitor. Also, the introduction of a new thread *T'* for each synchronized method places thread *T'* at the front of the entry queue. Therefore, the above rule allows the state exploration engine to explore the possibility that for each synchronized method, a new thread such as T' executing the method wins the competition over *T*, as *T* will enter the entry queue after the new thread when we execute the notify operation. Note that if a thread is inside the CS and the next operation is a notify operation, an *awaken* transition will be created for every thread in the condition queue (lines 7 and 8). These transitions explore the possibility that an awakened thread wins the competition over a new thread trying to enter the monitor (such as *T'*).

If both the entry queue and the CS are empty, *next_thread* will be null and this function returns (line 6). Otherwise, *next_thread* is the thread inside the CS or if no such thread exits the thread at the front of the entry queue. If the next operation to be performed by *next_thread* is a notify operation, a set of *awaken* transitions are created, one for each thread in the condition queue (line 7). Each of these *awaken* transitions, when executed, will awaken the associated thread. The reason for creating a set of *awaken* transitions is because a notify/notifyAll operation awakens an arbitrary thread in the condition queue. (Recall that according to Java's semantics, a notify operation does not necessarily preserve First-Come-First-Serve order.) For other types of operations, a single transition is created (line 8).

## 5. An Example Scenario

In this section, we use an example scenario to demonstrate our state exploration-based approach. Refer again to Fig. 2, which shows a Java monitor named *BoundedBuffer* that correctly solves the bounded buffer problem. In Fig. 2, a depositing thread *D* that enters the monitor first checks whether the buffer is full or not (line 3). If the buffer is full, thread *D* will execute wait (line 4) and will be blocked. Observe that wait is executed inside a while loop. Thus, when thread *D* is awakened at a later point, it will first re-check the loop condition *(fullSlots == capacity)* before it proceeds further. This re-checking of the loop condition is necessary since a notified thread does not immediately re-enter the monitor. Instead, it joins the entry queue and competes with other threads that are also trying to (re)enter the monitor. As a result, it is possible for another thread to barge ahead of thread *D*, making the loop condition *true* again before *D* is able to re-enter the monitor. Rechecking the condition ensures that *D* will wait again if this happens.

A subtle fault is introduced in monitor *BoundedBuffer* if the while-loop is replaced by an if-statement. This means that a depositing thread, after being notified, will proceed without re-checking whether the buffer is full or not. Below, we present a scenario that will be exercised and that will allow this fault to be detected when our state exploration-based approach is applied to the faulty version.

We will depict the scenario using a sequence of transitions between abstract states. For example, the first transition is

*[F, E, E, (N, N)] – introduce(D) → [F, E, D, (N, N)]*

In this transition, a pair of brackets represents an abstract state, and the arrow represents a transition that is labeled by the corresponding visible operation. The abstract states are created using the abstraction described in Section 4.1. To be specific, each abstract state consists of the abstract

```
 Set getEnabledTransitions () {
1.  let transitions be an empty set
2.  if (the entry queue is empty and either no thread
        is inside the CS or the next operation of the thread
        inside the CS is notify or notifyAll) {
3.    create an introduce transition for each synch. method
4.    add these transitions into transitions
    }
    else {
5.    let next_thread be the thread inside CS if exists
        or the thread at the front of the entry queue
6.    if (the next operation of next_thread is notify ) {
7.       create an awaken transition for every thread in
           the condition queue
      } else {
8.       create a transition for the next operation of
           next_thread
      }
9.    add these transitions into transitions
    }
10. return transitions;
  }
```

**Figure 4. Function *getEnabledTransitions***

values of four components, namely, *fullSlots*, the CS, the entry queue, and the condition queue, in the given order. The abstract values of each of these components are listed below:

- *fullSlots: U(nderflow)* if *fullSlots < 0*; *E(mpty)* if *fullSlots = 0*; *M(iddle)* if *0 < fullSlots < capacity*; *F(ull)* if *fullSlots = capacity*; and *O(verflow)* if *fullSlots > capacity*. Note that *U* and *O* are values that signify invalid states.
- *CS: E(mpty)* if no thread is inside the *CS*; *D(eposit)* (or *W(ithdraw)*) if a thread executing deposit (or withdraw) is inside the *CS*.
- *Entry queue*: *E(*mpty*)* if the queue is empty, *D(*eposit*)* (or *W(*ithdraw*)*) if the thread at the front of the queue is executing deposit (or withdraw).
- *Condition queue*: A type vector *(T, T')*, where *T* (or *T'*) is *N(ull)* if no thread in the queue executes deposit (or withdraw) and is *D(eposit)* (or *W(ithdraw)*) if there exists at least one thread in the queue that is executing deposit (or withdraw).

The complete scenario is as follows.

*[F, E, E, (N, N)] – introduce(D) → [F, E, D, (N, N)] – enter → [F, D, E, (N, N)] – wait → [F, E, E, (D, N)] – introduce(W) → [F, E, W, (D, N)] – enter → [M, W, E, (D, N)] – introduce(D) → [M, W, D, (D, N)] – notifyAll → [M, W, D, (N, N)] – exit → [M, E, D, (N, N)] – enter → [F, D, D, (N, N)] – exit → [F, E, D, (N, N)] – enter → [O, D, E, (N, N)] (An invalid state is entered)*

This scenario begins with an initial state, namely *[F, E, E, (N, N)]*, in which the buffer is full, and no thread is executing in the monitor. Since both the CS and the entry queue are empty, an *introduce* transition, i.e., *introduce(D),* is executed, which introduces a depositing thread *D1* into the entry queue and leads to state *[F, E, D, (N, N)]*. Next, thread *D1* enters the CS by executing an *enter* transition, which leads to state *[F, D, E, (N, N)]*. Since the buffer is full, thread *D1* executes a *wait* transition (line 4). As a result, *D1* is blocked on the condition queue, leading to state *[F, E, E, (D, N)]*. Since both the *CS* and the entry queue become empty again, another introduce transition, i.e., *introduce(W)*, is executed, which introduces a withdrawing thread *W1* into the entry queue and leads to state *[F, E, W, (D, N)]*. Similar to thread *D1*, thread *W1* enters the *CS* by executing an *enter* transition, which leads to state *[M, W, E, (D, N)]*. Note that since the buffer is not empty, thread *W1* does not enter the while-loop at line 10 and proceeds instead to withdraw an item. Thus, *fullSlots* is decremented by the execution of this *enter* transition. Immediately before *W1* executes notifyAll at line 15, an introduce transition, i.e., *introduce(D)*, is executed. This introduces another deposit thread *D2* into the entry queue and leads to state *[M, W, D, (D, N)]*. Next, thread *W1* executes an *awaken* transition, which awakens thread *D1*, leading to state *[M, W, D, (N, N)]*. Note that thread *D1* re-joins the entry queue and is placed after the second deposit thread *D2*. Then, *W1* exits the monitor, leading to state *[M, E, D, (N, N)]*. The next thread that enters the CS is thread *D2*. Since the buffer is no longer full, thread *D2* is able to deposit an item into the buffer, leading to state *[F, D, D, (N, N)]*. Next, *D2* exits the monitor, leading to state *[F, E, D, (N, N)]*. Thread *D1* then enters the CS. Since line 3 is now an *if*-statement, instead of a while-loop, thread *D1* proceeds without re-checking whether the buffer is full or not, leading to an invalid state *[O, D, E, (N, N)]*. This invalid state will be detected when thread *D1* exits, allowing us to detect the fault.

## 6. Case Studies

We built a prototype tool called *MonitorExplorer* that implements our state exploration-based approach. The tool consists of four major components: (1) an *exploration engine* that coordinates the entire state exploration process; (2) A *monitor driver* that hides from the exploration engine the specifics of how to communicate with the monitor under test; (3) A *monitor wrapper* that provides necessary runtime control for deterministic execution of the transitions; and (4) A *monitor toolbox* that simulates the Java monitor construct in a functionally equivalent manner. The monitor toolbox was adapted from [5] and allows us to access the internal state of a Java monitor, without having to modify the Java compiler or the Java virtual machine.

Before applying the tool, the user must provide an implementation of function *getAbstractState()*. In most cases, the user only needs to implement abstractions of data members in this function. The tool provides a built-in abstraction for the internal components of a Java monitor. In addition, the user can provide two optional functions, namely, *initialize*() and *evaluate()*. These two functions allow the user to provide their own initialization and evaluation code, respectively. Finally, the user needs to provide an input file that specifies the full path of the Java monitor class to be tested, and the parameters for invoking each synchronized method.

In the following, we present the results of applying *MonitorExplorer* to two classical textbook monitors, namely *BoundedBuffer* and *ReaderWriterSafe*, and a real-life monitor called *ArrayBlockingQueue* in the *java.util.concurrency* package in Java 1.5. All the results are obtained on a Windows desktop with 1GHZ CPU and 512 MB memory.

### 6.1 Monitor *BoundedBuffer*

The source code for monitor *BoundedBuffer* was given in Fig. 1. To conduct mutation-based testing, a set of mutants of this monitor were first created. Each mutant introduces a

single change to the original version and is intended to simulate a programming error. Then, we used our prototype tool to test each of these mutants. A mutant is said to be killed if our tool discovers a violation of any correctness requirement.

Two batches of mutants were used in this case study. The first batch contained 73 mutants that were created by a Java-based mutation tool called μJava [12][1]. μJava generates two types of mutants, one based on traditional mutation operators, e.g., changing a Boolean operator from ">=" to ">", and the other based on class-level mutation operators, e.g., changing an instance attribute to a static attribute. Since we are concerned with the synchronization behavior of a monitor, we only kept the first type of mutants created by μJava. There were 8 mutants in the second batch. These mutants were created based on several mutation operators that are unique to a Java monitor.

- If a *while* loop contains a *wait* operation, replace the *while* loop with an *if* statement.
- Replace a *notifyAll* operation with a *notify* operation.
- Remove a *wait*, *notify*, or *notifyAll* operation.

Recall that in order to apply our tool, the user needs to provide three functions, namely, *getAbstractState()*, *initialize()*, and *evaluate()*. In this study, function *getAbstractState()* implements the abstraction described in section 5. Function *initialize()* initializes a *BoundedBuffer* instance by depositing a given number of (random) integers into the buffer. Function *evaluate()* evaluates each state by checking the following conditions:

- The buffer never underflows or overflows.
- *# of completed withdraws <= # of completed deposits + # of initial items in the buffer.*
- If both the *entry* queue and the *CS* are empty, and if there exists at least one depositing thread in the condition queue, *# of initial items + # of completed deposits - # of completed withdraws = capacity.*
- If both the *entry* queue and the *CS* are empty, and if there exists at least one withdrawing thread in the condition queue, *# of initial items + # of completed deposits - # of completed withdraws = 0.*

Note that the last two conditions require both the entry queue and the CS to be empty. This ensures that all the existing threads that are not waiting in the condition queue have exited the monitor.

For each mutant, we first ran the prototype tool with the buffer initialized to be empty. If the mutant was killed by this run, we stopped testing; otherwise, we ran the prototype tool again with the buffer initialized to be full. A mutant that was not killed by either test run was said to be alive. Among the 81 mutants we created, 4 of them are functionally equivalent to the original monitor. Our prototype tool killed 57 mutants out of the remaining 77 mutants, making the ratio of killed mutants (over all the non-equivalent mutants) as 74.03%. We manually inspected each of the 20 mutants that were alive. All these mutants involved a single change to one of the following three attributes: *in*, *out*, *capacity*. The mutants involving changes to *in* and *out* were not killed because the changes did not affect the synchronization behavior. The mutants involving changes to *capacity* were not killed because our state abstraction scheme assumes that the capacity of a buffer is fixed, and some of these mutants cannot be killed by any program-based testing technique such as ours that only exercises feasible sequences. (These mutants can only be killed by showing that certain sequences cannot be exercised.) On the average, each test run took 1.36 seconds and explored 15.4 states and 24.7 transitions. Note that when we applied our tool to the original version of *BoundedBuffer*, it took 2.2 seconds and explored 33 states and 47 transitions.

## 6.2 Monitor *ReaderWriterSafe*

Monitor *ReaderWriterSafe* [11] solves the readers/writers problem, i.e., it allows multiple readers to read a shared variable at the same time, but requires writers to obtain mutually exclusive access to the variable. There are four synchronized methods, namely *acquireRead*, *releaseRead*, *acquireWrite*, and *releaseWrite,* defined in the monitor. A reader (writer) thread calls *acquireRead* (*acquireWrite*) before it starts reading (writing) and calls *releaseRead* (*releaseWrite*) after it finishes reading (writing). There are in total 34 lines of code in monitor *ReaderWriterSafe*. The source code of this monitor can be accessed at http://www-dse.doc.ic.ac.uk/concurrency/.

Similar to our previous study, we created two batches of mutants for monitor *ReaderWriterSafe*. The first batch contained 26 mutants and the second batch contained 7 mutants. In monitor *ReaderWriterSafe*, there are two data members, namely *readers* and *writing*, that can affect the synchronization behavior. Member *readers* is an integer variable used to keep track of the number of active readers, and member *writing* is a Boolean variable used to indicate whether there is an active writer. Function *getAbstractState*() implements the following abstraction. The value of *readers* is abstracted to *"-"* if *readers < 0, "0"* if *readers = 0, "R"* if *readers = 1,* and *"R+"* if *readers > 1*. The value of *writing* does not have to be abstracted, as it has only two values *true* and *false*. The internal components of the monitor are abstracted in the same way as described in section 5. There is no need for special initialization for this monitor, and thus function

*initialize()* is left empty. Function *evaluate()* evaluates each state by checking the following conditions:

- The abstract value of *readers* is never "-".
- *# of completed acquireWrite - # of completed releaseWrite <= 1*. (Intuitively, there can be at most one active writer at a given time.)
- If *# of completed acquireWrite - # of completed releaseWrite = 1, # of completed acquireRead - # of completed releaseRead* = 0. (Intuitively, if there is an active writer, there can be no active reader.)
- If *# of completed acquireRead - # of completed releaseRead* > 0, *# of completed acquireWrite - # of completed releaseWrite = 0.* (Intuitively, if there is an active reader, there can be no active writer.)
- If both the *entry* queue and the *CS* are empty, and if there exists at least one waiting writer, *# of completed acquireRead - # of completed releaseRead > 0*.
- If both the *entry* queue and the *CS* are empty, and if there exists at least one waiting reader in the condition queue, *# of completed acquireWrite - # of completed releaseWrite = 1*.

Again, in the last two conditions, the requirement that both the entry queue and the CS are empty ensures that all the existing threads that are not waiting in the condition queue have exited the monitor. Note that monitor *ReaderWriterSafe* has an implicit protocol that the four synchronized methods must be used in pairs. This constraint is enforced by the exploration engine by ensuring that *releaseRead* (or *releaseWrite)* is only executed by a thread that has already executed *acquireRead* (or *acquireWrite)*. We point out that if this constraint was not enforced, false negatives could be produced by our tool, as sequences that did not follow the implicit protocol could also be explored.

Among the 33 mutants we created, 3 of them are functionally equivalent to the original version. Our tool killed all the other mutants, making the ratio of killed mutants (over all the non-equivalent mutants) 96.67%. The only mutant that was not killed involved a change that caused method *acquireRead* to become read-only. Since no state change was detected after executing this method, the state exploration process terminated immediately. We note that this type of mutants can be killed by unit testing of individual methods, whereas our approach focuses on faults related to interactions between multiple methods. On average, each test run took about 1.99 seconds and explored 30.1 states and 43.7 transitions. When we applied our tool to the original version of *ReaderWriterSafe*, it took 3.65 seconds and explored 75 states and 106 transitions.

### 6.3 Monitor *ArrayBlockingQueue*

Monitor *ArrayBlockingQueue* can be considered as a real-life version of monitor *BoundedBuffer*. There are 18 synchronized methods and 758 lines of code in the monitor. The main purpose of our study of this monitor is to obtain some initial evidence about the scalability of our approach.

Before we report the results of this study, we make the following comments. First, unlike the default Java monitor, which only has an implicit condition queue, monitor *ArrayBlockingQueue* has multiple condition queues, a feature that is newly introduced in Java 1.5. Our prototype tool was extended to support multiple condition queues. Second, two methods in monitor *ArrayBlockingQueue* use a timed *wait* operation, which allows threads to wait for a specified amount of time. Our prototype tool currently does not support timed *wait*. Thus, the two methods were left out of our case study. Finally, among the 18 synchronized methods, 10 of them are query-only operations, and do not involve any *wait/notify* operations. These methods were not tested in this study since they did not affect the synchronization behavior of the other methods.

Table 1 reports the test results obtained for monitor *ArrayBlockingQueue*. Observe that the explored number of states and transitions actually decreases when the capacity of the queue is increased from 2 to greater than 2. Also observe that the number of explored states and transitions remains the same when the capacity of the queue is greater than 2. This can be explained as follows. Monitor *ArrayBlockingQueue* has a variable named *count* which keeps track of the number of items in the queue. Variable *count* is abstracted in the same way as variable *fullSlots* in section 5. Assume that the queue contains one item. Then, when we deposit one more item into the queue, we will get a new abstract state where the buffer becomes full if the capacity is 2. However, if the capacity is greater than 2, we will stay in the same abstract state where the buffer is neither full nor empty. According to algorithm *MonitorTest*, the current state will not be expanded if the corresponding abstract state has been visited before. As a result, when the capacity is greater than 2, the abstract state in which the buffer is full would not be reached when we start from an initial state where the buffer is empty. By the same token, the number of explored states as well as transitions will not change when the capacity is further increased. Note that when the capacity is greater than 2, the full state can be explored by initializing the buffer to be full.

| capacity | # of states | # of transes | # of paths | Time |
|---|---|---|---|---|
| 1 | 182 | 336 | 155 | 45.83s |
| 2 | 226 | 418 | 193 | 66.57s |
| >=3 | 80 | 140 | 61 | 10.96s |

**Table 1: Results for class *ArrayBlockingQueue***

## 7. Conclusion and Future Work

In this paper, we have described a state exploration-based approach to testing Java monitors. This approach can be used as a unit testing technique for concurrent programs. During state exploration, threads are introduced on-the-fly to simulate the race conditions that may occur when multiple threads call the same monitor simultaneously. It is our belief that many synchronization faults are caused by race conditions that are not accounted for. We wish to stress that our approach assumes that the methods in a Java monitor are properly identified as synchronized methods. Since mutual exclusion for individual synchronized methods is automatically enforced by the Java runtime, our approach focuses on detecting faults that are caused by improper and/or insufficient synchronization between different synchronization methods, i.e., synchronization involving wait/notify/notifyAll operations.

We are continuing our work in the following directions. First, we will try to remove the need for several assumptions that are currently made in our approach. For example, the synchronization behavior of a method is assumed to be independent from its arguments, and each method can be called independently. These assumptions restrict the applicability of our approach. Second, we will add necessary support for other synchronization-related features in Java, including timed wait, synchronization blocks, and multiple condition queues. Third, we will conduct a more thorough evaluation of our approach. In particular, we plan to compare the fault detection effectiveness of our approach to that of the approach reported in [10]. Fourth, we plan to add a graphical user interface to visualize the state exploration process. Such visualization will aid in understanding the behavior of a Java monitor and, in particular, will be of great help during the debugging process. Finally, we plan to extend our approach to non-Java monitors. In particular, we have implemented a monitor toolbox written in C++/Pthreads. We believe that the general framework presented in this paper can be adapted to test a monitor written in C++/Pthreads.

## Acknowledgement

We would like to thank Prof. Paul Strooper for sharing his work on testing Java monitors and Prof. Jeff Offutt for providing us with the µJava tool used in the case studies. We would also like to thank Weijia Deng for conducting one of the case studies.

## 8. REFERENCES

[1] T. Ball, R. Majumdar, T. Millstein, and S. K. Rajamani, "Automatic predicate abstraction of C programs," In Proc. the ACM SIGPLAN 2001 Conference on Programming Language Design *and Implementation*, pages 203–213, 2001.

[2] P. Brinch Hansen, "Reproducible testing of monitors," Software Practice and Experience, vol. 8, pp. 721-729, 1978.

[3] Richard H. Carver and Kuo-Chung Tai, "Replay and Testing for Concurrent Programs," IEEE Software, March 1991, pp. 66-74.

[4] R.-K. Doong and P. G. Frankl. The ASTOOT approach to testing object-oriented programs. ACM Trans. Softw. Eng. Methodol., 3(2):101–130, 1994.

[5] Richard H. Carver and Kuo-Chung Tai, *Modern Multithreading*, Wiley, 2005.

[6] J. Corbett, M. Dwyer, J. Hatcliff, C. P. Robby, S, Laubach, and H. Zheng. "Bandera: Extracting Finite-state Models from Java Source Code," *Proc. of the 22nd International Conference on Software Engineering*, June, 2000.

[7] P. Godefroid, "Model Checking for Programming Languages using VeriSoft," Proc. of the 24th ACM Symposium on Principles of Programming Languages, pp. 174-186, Paris, January 1997.

[8] K. Havelund and Tom Pressburger. "Model Checking Java Programs Using Java PathFinder," International Journal on Software Tools for Technology Transfer (STTT), 2(4): 366-381, April 2000.

[9] D. Kung, Y. Lu, N. Venugopalan, P. Hsia, Y. Toyoshima, C. Chen, J. Gao, ``Object state testing and fault analysis for reliable software systems**,"** Proc. of 7th International Symposium on Software Reliability Engineering, White Plains, New York, Oct. 30 - Nov. 2, 1996.

[10] B. Long, D. Hoffman, and P. Strooper, "Tool support for testing concurrent Java components", IEEE Trans. On Software Engineering, 29(6):555-566, 2003.

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

[12] Yu-Seung Ma, Jeff Offutt and Yong-Rae Kwon, "MuJava : An Automated Class Mutation System," *Journal of Software Testing, Verification and Reliability*, 15(2):97-133, June 2005.

[13] µJava Home Page, http://ise.gmu.edu/~ofut/mujava/.