A DYNAMIC FRAMEWORK FOR TESTING THE SYNCHRONIZATION BEHAVIOR OF JAVA MONITORS

by

ANDRES YANES

Presented to the Faculty of the Graduate School of

The University of Texas at Arlington in Partial Fulfillment

of the Requirements

for the Degree of

MASTER OF SCIENCE IN COMPUTER SCIENCE AND ENGINEERING

THE UNIVERSITY OF TEXAS AT ARLINGTON

May 2007

ACKNOWLEDGEMENTS

I would first like to thank my family and girlfriend for the incredible patience and understanding they have shown me while working on my thesis. Mom, Dad and Sergio, thanks for listening during the countless conversations regarding my research and presentation. You helped me immensely in working out many of the ideas and problems that I encountered even though you felt you knew little about what I was talking about. Sudha, thank you for your support and encouragement, even when this process has required me to reprioritize my time. You have been a driving force that moved me forward without hesitation.

I would also like to thank my supervising professor Dr. Yu Lei who has seen me through this thesis. I have enjoyed all our talks and appreciate all the extra time you put in during last minute meetings and phone conversations. I have learned a lot from your example.

My thanks also go out to the CSE department staff and faculty for their efforts and guidance during the completion of this degree. In particular I would like to thank Ms. Camille Costabile who answered numerous questions with a smile and always kept me on track.

Lastly, I would like to thank all my friends who have supported me with their prayers and best wishes.

April 20, 2007

ABSTRACT

A DYNAMIC FRAMEWORK FOR TESTING THE SYNCHRONIZATION BEHAVIOR OF JAVA MONITORS

Publication No. _____

Andres Yanes, M.S.

The University of Texas at Arlington, 2007

Supervising Professor: Dr. Yu Lei

A Java monitor is a specialized class that is used to synchronize the behavior of threads in a Java program. The monitors in a Java program must be adequately tested to ensure the correctness of the program. In this thesis we propose a dynamic framework in which a Java monitor is tested by exploring its state space in a depth-first manner. The state exploration procedure consists of dynamically creating method sequences to exercise the possible synchronization behavior of the monitor. During exploration, new threads will be created on the fly to simulate different scenarios that result from threads reaching the monitor at different times. Each state reached is represented by a collection of data members that have been identified as having an affect the synchronization behavior of the monitor as well as an abstraction of the thread states.

A prototype tool that implements our framework has been built and has been used to evaluate the effectiveness of our approach in five case studies. In each case study, mutations to the original source code of a Java monitor are introduced to create variants that represent common mistakes made by programmers. The experimental results show that our framework is effective in detecting the synchronization failures in the case studies.

TABLE OF CONTENTS

LIST OF ILLUSTRATIONS

LIST OF TABLES

CHAPTER 1

INTRODUCTION

Today, a large number of software solutions are multi-threaded. Many of our desktop applications, such as word processors and web browsers, require multiple tasks executing concurrently to implement a seamless solution. Company services, such as purchasing or scheduling, use web-based, multi-tier solutions that must scale to allow millions of simultaneous users to perform transactions that access shared data.

Though multi-threading enables the creation of complex systems, it introduces complexities in their development. When multiple threads are executed within a system, the execution order and time allotted for each is non-deterministic. As a result, they may display different behaviors from execution to execution. This non-determinism must be managed by means of thread synchronization to ensure that threads behave as expected.

Thread synchronization can be grouped into two general categories, mutual exclusion and condition synchronization [33]. Mutual exclusion is used to ensure that when two or more threads are attempting to access shared data each thread's access operation is atomic. One way to implement this behavior is by means of a critical section. A critical section is a block of code that can only be executed by a single thread at any given time. Condition synchronization is used when a thread should only be allowed to proceed if a specified condition has been satisfied.

1.1 Java Monitors

The Java language [14][39] uses a monitor-based [18] approach to thread synchronization. A Java monitor is a specialized class that is used to encapsulate application specific thread synchronization logic in a Java program. This class consists of one or more synchronized methods, each of which is a critical section of the monitor and guarded by a single lock object. This lock object is implicitly acquired and released each time a thread enters and exits a critical section. A model of a Java monitor can be seen in Figure 1.1.

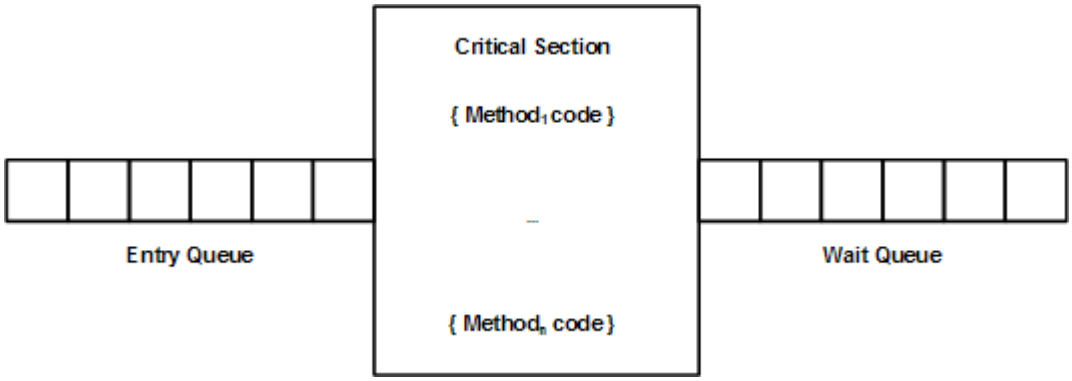

Figure 1.1 Model of a Java Monitor

Once a thread is executing in a synchronized method of the monitor, all other threads attempting to execute a synchronized method must wait in the monitor's entry. A thread that has entered a critical section, may exit the critical section either by successful completion of its method or by making a call to one of the *wait* primitives. If a *wait* primitive is called, the thread must release its lock after which it is moved to the monitor's wait queue. After a thread releases its lock and exits the critical section, a

thread is selected from the entry queue, allowing it to acquire the lock and enter the critical section.

For a threads to exit the wait queue, another thread executing in a critical section must make a *notify* or *notifyAll* primitive call. If the wait queue is not empty at the time these calls are made, one or more threads are moved from the wait queue to the entry queue before execution resumes in the calling thread. This type of signaling discipline is known as signal-and-continue. The only difference between these primitives is that the *notify* call will result in the selection of a random thread from the wait queue, while a *notifyAll* call affects all threads in the wait queue. It is also possible for a thread to exit the wait queue if it called a *wait* primitive with a timeout argument. In this event, if the thread is still waiting after the timeout period has expired, it will be moved to the entry queue. We will not address this type of *wait* primitive in our approach.

1.2 Testing Java Monitors

Unlike other Java classes, Java monitors are intended to be accessed by multiple threads at the same time. Though these threads will reach the monitor's entry queue in a specific order, this ordering is non-deterministic. In addition, monitors are passive objects that have no inherent thread of execution. These two properties make testing Java monitors difficult. To verify a monitor's synchronization behavior, test cases consisting of a sequence of method calls must be created that simulate possible thread interaction with the monitor.

Test case creation for this problem domain is conceptually challenging. The overall goal is to create method call sequences that collectively provide sufficient synchronization coverage to allow the tester to verify that the monitor's implementation enforces its expected behavior. However, to determine the level of coverage needed, test cases must address the following issues. Consider that a test case will start with one thread that will be used to call each method in the test sequence. This will continue until execution reaches a *wait* primitive that results in the thread being moved to the wait queue. Once this occurs, a new thread must be created to continue the test case sequence.

This behavior is expected, as monitors are designed to allow threads to proceed within a synchronized method as long as certain conditions are satisfied. If one or more of these conditions are not satisfied, the thread must wait until another thread changes the state of the monitor and satisfies the condition. In complex monitors, conditions may be spread across multiple methods whose combinations can only be exercised with multiple threads. In these cases, some synchronization faults can only be detected with a certain minimum number of threads. So an important question is, for a given monitor, how many threads will be needed to provide sufficient coverage?

In addition, once a thread is awoken and allowed to re-enter the critical section, it is not guaranteed to find the conditions it was waiting on to still be satisfied. This is due to the fact that an awakened thread does not re-start execution immediately. Instead, it has to compete with other threads to re-enter the critical section. Therefore, it is possible that one or more threads could enter the critical section and falsify the

condition, before the awoken thread is able to re-enter the critical section. In an open system, there are an infinite number of possible sequences that can occur before an awoken thread is allowed to re-enter, each having the potential to affect the synchronization behavior. Given this, how many test sequences combinations must be created to cover the effect of this non-determinism on the synchronization of both new and awoken threads?

In this thesis we present a framework that dynamically creates and executes test sequences that explore the state-space of a Java monitor. Each test sequence is created according to a systematic process that guarantees reproducibility across multiple test executions of the same monitor. This requires the framework to override the Java scheduler behavior as well as the *notify*, *notifyAll* and *wait* primitives. To consider all possible competing threads that could gain entry into the monitor, the framework will compile a list of candidates that consists of each monitor method as well as any awakened threads. The framework then selects the first candidate and allows it to enter the critical section. This process will repeat at each state reached until either the current state has been previously visited or a violation of the monitor's expected synchronization behavior is detected. If a previous state is reached, the current sequence is terminated and the framework will backtrack to the previous state. After a backtrack, the scheduler will retrieve the candidate list for the current state and select the next candidate for execution. If execution of a test sequence requires more threads than are currently available, new threads will be created on the fly. In this manner, the test sequences are created according to a depth-first search of competing candidate threads,

each exploring one aspects of the synchronization behavior that could be exercised by the monitor.

A major concern of this type of approach is the state-space explosion problem. In our framework, the monitor state is represented by the value of some selected data and an abstraction of select thread states. The threads used in the thread abstraction consist of those found in the wait queue as well as any threads attempting to re-enter the critical section after being moved from the wait queue to the entry queue. The abstraction groups threads according to the last wait primitive executed, and assigns each group an abstract value. If a group consists of threads currently in the wait queue, the assigned value is WAITING, while a group consisting of threads found in the entry queue is assigned the value ENTERING. If a group consists of threads found in both queues, the value assigned is BOTH. Using this abstraction reduces the granularity of the thread information, resulting in fewer states to be visited. In addition, the framework incorporates a configuration feature that allows sequences to be terminated when specific values or range of values have been reached. This feature can be used to reduce the number of states visited as well as restrict the sequencing of method calls that require a specific ordering.

A mutation-based approach was used to evaluate this framework using five monitor case studies. In each case study, mutants were created by applying one of two categories of mutation to the original source code. The μJava [34] tool was used to create mutants that introduce mutations representative of traditional mistakes made by programmers. In addition, mutants were created which change the behavior of the

synchronization primitives used in the monitor. The results gathered from executing these mutants show that the framework is effective in detecting these mutations.

Our contributions in this thesis are as follows:

- We have created a new approach based on the dynamic framework of the MonitorExplorer tool [29]. The approach redefines states, transitions and rules used to introduce new threads during the exploration of a monitor's state space. As a result, we have increased the state space and number of paths that can be visited during each monitor test.
- We have created a prototype tool that implements the approach.
- We have performed an initial evaluation of our approach using our prototype tool.

1.3 Thesis Outline

Chapter 1 introduces the topic of concurrent programming and thread synchronization. A conceptual model of a Java monitor is then given followed by an explanation of a monitor's behavior as threads interact with its synchronized methods. The difficulties in testing a Java monitor are then discussed followed by a description of how our framework approaches testing a Java monitor.

Chapter 2 discusses other related work in the areas of monitor testing and state-space exploration. This discussion opens with a proposed methodology for analyzing monitor precondition, creating test cases to exercise the preconditions and execution of the test cases. Several approach are then described that use this methodology in several languages. These approaches use a framework that allows for deterministic execution.

In addition, a framework similar to the one proposed in this thesis is described. The discussion ends with an overview of testing approaches that have used state-space exploration for testing concurrent components.

Chapter 3 details the state-space exploration approach used in our framework to test Java monitors. This begins with a discussion of core concepts used in our state-space exploration and how they are used to configure the framework to detect application specific properties of the monitor being tested. This is followed by a detailed description of the state-space algorithm that shows how the core concepts described above are used to create the dynamic test sequences that explore the monitor synchronization behavior. An example is then presented using a bounded buffer monitor that has been modified to contain a synchronization fault. This example shows how the fault is detected through the exploration of the monitor's state-space using our approach.

Chapter 4 describes the design of the prototype tool created to implement our approach. The first section describes the interfaces defined in the *UserProvided* package. These interfaces must be implemented by the tester and provide the application specific components used during the state-space exploration. The next section describes the class and event interface defined in the *Hook* package. This package is used by the framework to intercept *notify*, *notifyAll* and *wait* primitive calls. The last section provides a description of the high level classes that make up the prototype tool framework. The description includes the steps a tester must take to initialize the framework as well as the steps required to execute a monitor test.

Chapter 5 discuses the experimental design used to evaluate the prototype tool. The mutation-based approach used is described as well as the different type of mutants created. A description is given of the hardware used to test each mutant. The results for each of the five monitors used in our case studies are then described with a summary of their expected synchronization behavior and a presentation and analysis of the their results.

Chapter 6 provides a summary of the framework current abilities and limitations. Based on the observations from each case study performed, several goals for future work are proposed that include using an abstractor for the data members in the state representation, extending the framework to support the full set of synchronization primitives supported by Java 1.5 and further evaluation of our approach that focuses on performance and the effectiveness in detecting synchronization failures in more complex monitors.

CHAPTER 2

RELATED WORK

In this section we will review research related to testing monitors and concurrent programs. We will open with general approaches that can be used to test monitors and follow up with a discussion of those approaches specifically designed to test monitors implemented in Java. This will be followed by a review of approaches that use state-space exploration to test the design and implementation of concurrent programs and ends with an overview of tool support that can be used to verify varying degrees of Java synchronization.

In [17], Hansen introduced a systematic methodology for testing Pascal Monitors. This methodology involves four steps. The first is to inspect each monitor method to determine a set of preconditions that will exercise each path in the method at least once. Next a test sequence consisting of monitor method calls is created that exercises each precondition identified in the previous step at least one time. The third step is to create a driver to execute the test case using threads to execute each method called in the sequence. To ensure correct ordering of threads in the driver, a clock monitor is used that synchronizes the threads. The last step is to execute the driver and compare the output with the expect output of the monitor.

Carver and Tai presented an approach in [3][4], which generalizes Hansen's methodology to allow for deterministic execution and replay of concurrent programs

using either an implementation-based or language-based tool. Test sequences are created based on feasible synchronization sequences that are representative of the synchronization constructs available in the programming language begin used. Using these sequences, the test program is transformed by the tool to ensure the correct sequence of events. In addition, during normal execution of a concurrent program synchronization sequences are recorded that capture the synchronization events that occur. Using these sequences, the execution can be replayed to allow for debugging or regression testing. Examples of how this approach could be used to testing monitors and semaphores were given.

Hansen's methodology was also adapted to test Java monitors [19][32][42]. Noting the differences between Java and Pascal synchronization constructs, the preconditions identified in the first step of Hansen's methodology was extended to ensure loop coverage and consideration of the effect of signaling during execution for different numbers and types of threads in the wait queue. The reason for the first extension is based on the fact that since Java does not provide multiple condition queues, wait primitives are usually put inside a loop construct so that a thread reentering the critical section can verify that the condition it was waiting on is still satisfied. A tool called ConAn was also provided to automate the third and fourth steps based on the test sequence calls identified from the first and second steps.

To assist in creating test sequences an approach was presented in [28] that requires a formal specification using extended UML state diagrams that captures the system functionality and the concurrency behavior. Using this specification, state

machines are generated via model checking that can then be transformed into test sequence usable by tools such as ConAn.

In each of these approaches, the first two steps require manual analysis of the monitor to derive the preconditions and test cases. This process can be time consuming and produce incomplete or incorrect results if the tester does not fully identify the necessary preconditions or introduces errors in the test created. Because our approach dynamically creates test sequences by simulating different thread combinations that can be executed as a monitor is used, this potential for user error is minimized and test cases can consider more than one possible sequence of method calls that can occur. This is important as synchronization failures will often times occur because the implementation of a monitor did not consider a combination of thread interactions that may occur due to the non-deterministic arrival of threads to the entry queue.

An alternative tool called MonitorExplorer was described in [29]. This tool uses a state-space exploration based approach to dynamically evaluate different execution sequences through a monitor. This tool allows for monitors to be tested in isolation and allows a user to define synchronization behavior properties that must be verified at each state reached during exploration. A unique property of this approach is that threads are created on the fly as needed to support the simulation of different types threads trying to compete in the entry queue for entry into the critical section. The approach described in this thesis uses this same framework to test a Java monitor and we would like to ensure that full credit is given for its novel approach. States visited during exploration are represented using data member and wait and entry queue abstraction. Using this state

representation allows each path explored to be bound by the constraints introduced in the abstraction. There are three differences between this approach and the one we present. The first is the manner in which sates and transitions are defined. The second is the state representation used to capture each state and the last is the rules used to introduce new threads. These differences are discussed in detail in Chapter 5.

Other state-space exploration approaches have been presented for testing concurrent programs. Model checking is one such type of approach that is used to analyzing the correctness of a program and traditionally is performed at the design level. Using a design or requirements for a program, a formal abstract model is created that is a simplified representation of a specific part of the program being evaluated. These models are often times based on finite state machines or call graphs [5][6] that express aspects of the design's control flow or data flow. Using this model and a specification that defines a set of well-defined properties of the model most commonly expressed in temporal logic as input, tools such as SPIN [22][23] can be used to explore the model's state-space and evaluate the properties. Success in using this type of approach is often dependent on creating a model that is accurate in its representation and in the ability to express the properties of the model sufficiently. Different model checking methods have been proposed such as [30][40][37], which have increased the range and type of properties that can be expressed using this type of approach.

As mentioned, these type of approach can be used to verify the property of concurrent programs at the design level, however recent efforts have also focused on extracting models based on the implementation code. The JCAT tool [9] was the result

of an effort directed at translating Java source code to Promela so a program's implementation can be verified using SPIN. The tool incorporates several straightforward techniques to reduce state complexity and state explosion such as user provided annotations to reduce the number of variables defined in a state, static analysis to eliminate unused or redundant resources and atomic blocks around local variable access to reduce the number of states visited. Synchronization constructs are simplified by using template abstractions that still preserve the accuracy of the model. The Bandera toolset [8] also has a similar goal of extracting an abstract model from Java code. This tool, however, approaches model creation from a different direction by creating a new model for each property specified by the user as annotations in the code using the Bandera Specification Language. This approach allows the tool to optimize each model using abstract interpretation and program slicing to identify only those parts of the implementation that are necessary to verify the specified property. The created models and specifications can then be returned in one of several popular model-checking languages. The first generation of Java PathFinder [21] is a tool also designed to convert Java source to Promela models. Assertions are provided to the tool for verification using annotation in the code. Though this tool currently only supports a subset of the Java language, it can model dynamically created objects, threads, synchronization constructs, exceptions as well as a significant amount of language constructs. In addition, deadlock detection and an abstraction workbench are also incorporated within the tool. No techniques are applied to the transformation phase that

reduce the model size, therefore the Java code under test must have a finite state-space and be intended more for unit testing.

The second generation of Java PathFinder [25][41] takes a new direction on testing Java programs by using explicit state model checking supported by a custom JVM that interprets byte code instead of having to analyze source code. Several techniques have been added to the tool to address the state explosion problem and allow the tool to handle large or infinite state spaces. State compression and predicate abstraction using the Stanford Validity Checker [2], combined with annotations of user-provided predicates are used to reduce the state complexity and size. In addition, partial order reductions guided by slicing information gather using the Bandera toolset are used to minimize the state-space. Runtime analysis is also included to detect data races using the Eraser algorithm [38] and deadlock using the LockTree algorithm [41]. Currently the tool does not support temporal logic model checking, however, support is planned for the future. Another tool that incorporates explicit state model checking is Verisoft [13]. This tool uses a state-less search algorithm that incorporates sleep sets and persistent sets to address state explosion and reduce the number transitions executed. Using this approach deadlocks and assertion violations can be detected.

Several tools have also been presented that focus on testing Java at the implementation level using purely static analysis techniques. The Jlint tool [1][26] performs a global control flow analysis and local data flow analysis of Java byte code to identify faults in a Java program. Support for multithreaded code is provided in the analysis by means of call graphs and accessor dependency graphs but is limited to

detection of race conditions and deadlock. The use of this tool requires no special configuration from the user and is designed to be quick and efficient. A similarly easy to use tool is FindBugs. The FindBugs tool [12][24] performs an analysis of Java byte code and detects possible errors based on recognized bug patterns that represent common mistakes made when implementing solutions in Java. The collection of patterns as well as the static analysis performed by the tool includes support for detecting possible errors related to multithreaded programs. The types of synchronization errors that can be identified are related to general detection of possible deadlock, race conditions and conditional synchronization. As a result, the results generated by the tool must be analyzed by the user to evaluate whether any identified patterns corresponds to an implementation fault and thus requires time and knowledge of the software tested. The ESC/Java2 tool [7][11] combines static analysis and theorem proving approaches to test a Java program. In addition to detecting common runtime errors from a static analysis of the code, JML [27] based annotations, known as pragmas, can be inserted into the code to customize the type of checking performed during the analysis. These pragmas are verified using the Simplify [10] theorem prover that is incorporated into the tool. Similar to the other tools, the support for synchronization related faults is limited to general detection of deadlock and race conditions. Theorem provers, such as [15][35][36], have been successfully used to formally verify concurrent programs. This verification is typically performed at the design level by through the use of a formal specification however limited tool support has been added to support a transformation of code to a formal specification that can be

evaluated by the prover. The formal specification, typically defined using a first order logic language such as UNITY, is used by these provers to verify properties of the system and is focused on what a program does and not how it does it.

These implementation based approaches are limited by the fact that each tool requires the concurrent program tested to be a closed system. Therefore to use these tools to test a monitor would require the creation of a driver to interact with the monitor that defines not only the order of method executions but also the types and number of threads that would used. This differs from the approach presented in this thesis in that a Java monitor can be tested in isolation due to the fact that threads are created on the fly to simulate different combinations of thread entry into the monitor's critical section.

CHAPTER 3

APPROACH

In the following chapter we will detail the state-space exploration approach used to test Java Monitors. The chapter consists of three parts. The first part will cover why and how state-space concepts have been adapted to test Java monitors. The second part will detail how the search algorithm ties these components together. In the last part we will present an example that shows the execution of the state-space algorithm using a bounded buffer monitor.

3.1 Core Concepts

The success of any state-space exploration approach is determined by whether an implementation's level of coverage is capable of meeting the needs of its designer. In this case, our goal was to provide adequate synchronization coverage so that in a best-case scenario, all synchronization failures of a user provided Java monitor are exposed.

A key difficulty with this problem domain is due to the fact that each Java monitor serves an application specific need, of which the framework has no knowledge. To overcome this problem, our state-space algorithm must be tailored according to each monitor's application specific qualities.

In the rest of this thesis we will define the following terminology to generalize the key aspects of the approach:

Global Signaling Primitive – any synchronization primitive that wakes up all threads in the wait queue and moves them to the entry queue.

Random Signaling Primitive – any synchronization primitive that wakes up a random thread from the wait queue and moves it to the entry queue.

Re-entry Thread – a thread that has been woken up and moved to the entry queue but has not yet been allowed to re-enter the critical section.

Waiting Primitive – any synchronization primitive that results in a thread releasing its lock, exiting the critical section and being moved to the wait queue.

3.1.1 State Representation

The core of any state-space based approach centers on how the state, which represents a snapshot of select data at each stage of execution, will be represented. The reason for this is that in most state-space implementations, all decisions are based in some part on the states gathered during execution. For this reason, the techniques used to select and represent the data that make up the state play a significant role in the success of the algorithm.

Our state representation can be broken down into two parts. The first part is composed of a collection of monitor data members. These members are identified by the tester as having an effect on the synchronization behavior of the monitor. Each time the monitor state is captured, the current value for each member is retrieved and stored. This method of representation requires minimal analysis and interaction from the tester. It may, however, result in a state-space explosion if one or more data members selected by the tester increases or decreases without bound. To address this issue, the framework

provides a feature, described in Section 3.1.4, which allows the tester to limit the number of states that can be visited.

The second part of the state representation consists of data that abstracts the state of waiting and re-entry threads. This abstraction groups threads according to the last waiting primitive they reached and assigns each group a value based on their collective state. If a group consists exclusively of waiting threads, the assigned value is WAITING. A group consisting exclusively of re-entry threads is assigned a value of ENTERING. And if a group contains both waiting and re-entry threads, the assigned value is BOTH.

A special note must be made here regarding the effect a random primitive call has on the state of threads in the wait queue. A goal of the framework is to simulate all execution sequences that result from threads entering the entry queue at different times. When a random primitive call is made, all threads in the wait queue become potential candidates for re-entry. Because of this, the framework must consider the scenarios that result from the selection of each candidate thread entering the monitor at different points during an execution sequence. To enforce this behavior, the framework will delay the affect of the random primitive call (see Section 3.2.2 for details) that can introduce ambiguity into the thread abstraction.

As an example, consider a monitor with a wait queue containing four threads: T_1, T_2, T_3, T_4. Threads T_1 and T_2 enter the wait queue from the same waiting primitive making them one group, which we will call G_1. T_3 and T_4 enter the wait queue from a different waiting primitive making them a second group, which we will call G_2. Since

all four threads are waiting, the value of G_1 and G_2 in the thread abstraction would be WAITING. Now consider that a method executing in the critical section executes two random primitive calls. This will result in two random threads being moved from the wait queue to the entry queue, thus affecting the thread abstraction. If T_1 and T_2 are chosen the abstract value of G_1 will become ENTERING. Likewise if T_3 and T_4 are chosen, the abstract value of G_2 will become ENTERING. If, however, one thread is chosen from each group, G_1 and G_2 will have an abstract value of BOTH. Because of this, the abstract value of both groups cannot be determined until thread selection occurs. In order to remove this ambiguity, the framework considers all threads that could be selected due to a random primitive call as ENTERING until selection of a thread has occurred.

3.1.2 Monitor Initialization

During execution, it will become necessary to create instances of the monitor under test. Even though the framework will have access to the monitor Java class, creation and initialization of the monitor ultimately affects the direction of the test and therefore has been delegated to the tester.

The framework requires that the tester provide a Java class, which we will call the *MonitorInitialization* class, which is responsible for creating instances of the monitor under test. This class consists of two methods; one to report the total number of unique monitor instances that can be returned by the provided class and the other to return a monitor instance given a monitor instance number. Combined, this

functionality provides the tester with the means to automate the testing of several test cases with a single execution of the framework.

3.1.3 Methods and Method Arguments

In addition to supplying monitor instances, the framework requires the tester to identify the synchronized monitor methods that will be used during the state-space exploration. Each method must be identifiable by a unique method ID. Methods containing parameters are of special interest. Because method arguments may affect the synchronization behavior of the monitor, the framework cannot be responsible for determining what arguments should be used. In addition, one or more of the arguments may be a non-standard complex data type that cannot be created and/or initialized by the framework.

For this reason the tester must provide a Java class, which we will call the *MethodArguments* class, that is responsible for providing arguments for all synchronized methods used during execution. This class consists of two methods. The first method accepts a unique method identifier and returns the number of argument instances returned by the implementation for the specified method. The second method accepts a unique method identifier and an argument instance number and returns an array of Objects. Each member of the array must be re-castable to the expected parameter type for the given method identifier.

```
public 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 e){}}

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 e){}}

12.     value = buffer[out];
13.     out = (out + 1) % capacity;

14.     if (fullSlots-- == capacity)
15.        notifyAll();

16.     return value;
   }
}
```

Figure 3.1 The *BoundedBuffer* Monitor

As an example, let's return to the *BoundedBuffer* example in Figure 3.1. The *withdraw* method has no parameters and thus would need no arguments when it is

called. The *deposit* method, however, has an integer parameter which the framework must supply each time the method is called. Because the framework does not understand the application specific nature of the monitor, it cannot determine how to configure the *deposit* method arguments. Therefore to initialize a test of the monitor, the *MethodArguments* class instance would be created. Using this object, the framework would query for the number of *deposit* argument instances supplied by the tester. Then for each instance, a new *deposit*/argument combination would be added to the collection of synchronized methods called during execution. A single argument instance is sufficient in this situation because the synchronization behavior of the monitor is not dependent on the value of the deposit argument. Therefore the *MethodArguments* object could simply return an argument instance count of 1 for the *deposit* method that would correspond to an Object array containing some integer value. If the tester decided that they would like to consider 3 arguments for the *deposit* method, they would simply need to return 3 as the argument instance count and provide an Object array containing a single integer argument for each argument instance, such as 1, 2 and 3. Given this scenario, the monitor would be configured to execute with 4 method calls: *withdraw*(), *deposit*(1), *deposit*(2) and *deposit*(3).

3.1.4 Application Specific Property Checking

During execution, each monitor's synchronization behavior may be subject to some properties that must be verified to detect any application specific synchronization failures. Because the application specific behavior of the monitor is unknown to the framework, it requires guidance to detect these potential synchronization failures.

To overcome this limitation, the framework requires that the tester provide a Java class, which we will call the *PropertyChecker* class, with a single method that will be queried at each state reached during the state-space exploration. This method will have access to two aspects of the monitor under test; the current state and information regarding completed and waiting threads. Using this information, the method must evaluate the application specific properties of the monitor. If a failure has been detected, the method must throw a property violation exception. If no failure has occurred, the method may return a TRUE and execution will continue.

Because the current implementation does not use an abstraction for the data members stored in each captured state, it is possible that there may be one or more data members that could increase or decrease in value without bound. In addition, it may be necessary to restrict the sequencing of method calls that require a specific ordering. To overcome these issues, this method can also be used to terminate the current sequence by returning a FALSE value.

It should be noted that the intent of this method is to allow the tester to detect application specific failures using information regarding the expected behavior of the monitor rather than implementation specific data. There are two key reasons why this is advisable. The first is that to verify the expected behavior based on the implementation will require the tester to analyze the underlying code of the monitor since implementation details are not typically included in software requirements or specifications. If the defined properties resulting from this analysis are incorrect or incomplete, the results of the test may be inaccurate. Secondly, it is possible that the

implementation may change during the lifetime of the monitor. If the tester is unaware of these changes, the synchronization behavior properties defined using implementation details may become insufficient or inaccurate to verify the monitor's behavior during execution.

Using a combination of completed and waiting thread information, many monitor properties can be computed in an implementation-independent manner. As an example, consider that a bounded buffer monitor must not be able to deposit more than the maximum capacity of the buffer. One way to approach encoding this property would be to state that 0 <= *fullSlots* <= *capacity*. Although this would indicate that no more deposits were made than the maximum allowable, it is linked to the current implementation of the monitor. As an alternative, the same property could be verified by checking that the number of filled buffer slots when the monitor was initialized + number of completed deposits – number of completed withdraws <= *capacity*. By encoding the property in this manner we are still ensuring that at each state that the number of deposited items is less than the buffer capacity, however, the property is solely dependent on the synchronization behavior of the monitor rather than its implementation.

3.2 The Algorithm

In the following section, we will describe the algorithm used during execution in two parts. The first part will describe the overall structure as well as detail the purpose and outcome of each action. The second part will provide details regarding the framework's thread selection and execution.

3.2.1 Execution Framework

The algorithm used by the execution framework is shown in Figure 3.2. To begin execution, the *Test* method must be called and the framework must know the location of the user provided components. Optionally, a Maximum Consecutive Duplicates Allow (MCDA) (see Section 3.2.1.1 for details) value may be supplied to override the default value of zero.

Execution of the *Test* method begins by initializing the framework as seen in lines 1 through 6. The first two operations are responsible for loading the user provided components described in Sections 3.1.2 and 3.1.4. The third operation uses components described in Section 3.1.3 to create the list of synchronized methods that will be used during execution. The last three operations initialize data structures used during the state-space exploration. *Executioncontexts* is a stack that will be used to maintain *ExecutionContext* instances for each state reached along the current execution sequence. *States* is an ordered list that will maintain a collection of reached states and *context* is an instance of the complex data structure *ExecutionContext* that is used by the framework to select and execute threads and will be explained in Section 3.2.2.

```
Initialize:
1. create a instance monitorinitialization of MonitorInitialization
2. create a instance propertychecker of PropertyChecker
3. let executablemethods be a list of synchronized method
4. let executioncontexts be an empty stack
5. let states be an empty list
6. let context = a ExecutionContext data structure that contains a candidate
        threads list and a pointer to the next candidate to execute

Test(){
7.   For monitorid = 1 to monitorinitialization.count()
    {
8.      monitor = monitorinitialization.getMonitor(monitorid)
9.      SaveState()
10.     context = a new ExecutionContext

11.     do
        {
12.             while (Execute()){
13.                 if (ContinueSequence() AND propertychecker.check()){
14.                     SaveState()
15.                     executioncontexts.push(context)
16.                     context = a new ExecutionContext
                    }
                    else
17.                     Backtrack()
                }
18.             Backtrack()
19.             context = executioncontexts.pop()
20.             let lastState = states.removeFromHead()
21              states.addToFoot(lastState)
22.     }while (context != null)}
    }
SaveState(){
23.   let state = getState()
24.   states.addToHead(state)
    }
```

Figure 3.2 The State-Space Exploration Algorithm

Once the framework is initialized, the *Test* method will begin execution at the outermost loop of this algorithm, seen on line 7. Each iteration of this loop will begin a new exploration of the monitor's state-space using a different initialized monitor as provided by the tester. For each monitor instance available, the algorithm will initialize the test run by acquiring a *monitor* instance, calling the *SaveState* method to save the initial state of the monitor and create a new *executioncontext*, as seen in lines 8 and 10. Each time the *SaveState* method is called, the current state of the monitor will be captured according to the state representation described in Section 3.1.1. This *state* will then be added to the head of the *states* list.

Lines 11 through 22 are responsible for driving the exploration of the monitor's state-space. Each iteration of the inner loop on line 12 selects a thread and allows it to enter the critical section. Once the thread exits the critical section, the monitor is considered to have transitioned to a new state that will be captured and evaluated by the framework. If the state has not been previously visited and is accepted by the *propertychecker*, the loop will execute another thread thus propagating the current branch of the state-space. Otherwise, the current branch is terminated and the exploration will backtrack to the previous state.

Execution of this loop results in a depth-first exploration of each branch of the monitor's state-space and will continue to iterate until a time is reached when all possible candidate threads for the current state have been executed. When this occurs, the exploration will attempt to backtrack to the prior state in the current branch of the execution sequence. In the event that state prior to the backtrack was the initial state

where execution began, the backtrack will not be able to restore a prior state, since it will not exist. When this occurs, the exploration of the current monitor instance is complete and must be terminated. The loop condition on line 22 is responsible for detecting this event. Let us now take a detailed look at how this is behavior is implemented by the framework.

As described previously, the *Execute* method loop on line 12 will select the next candidate thread, execute it and block until the synchronized method it is executing completes. Once the thread has finished execution of the method, the *Execute* method will return a TRUE value indicating that a transition has occurred. In this case, the newly reached state must be evaluated to determine if it the current path will continue or be terminated.

Both the framework and the propertychecker are queried to make this determination. As seen on line 13, the *ContinueSequence* method is called first to verify if the framework will allow the current path to continue. This method can be seen in Figure 3.3.

```
ContinueSequence(){
1.  let MCDA = the user provided MCDA value or 0 if no value provided
2.  let state = getState()
3.  let result = NOT (states.contains(state)

4.  if (result == FALSE){
5.      let duplicatestates = number of consecutive states identical to state,
                              starting from the head of the states list
6.      result = duplicatestates > 0 AND duplicatestates <= MCDA
        }

7.  return result
  }
```

Figure 3.3 The *ContinueSequence* Method

This method begins by getting the *MCDA* value and current *state* instance as seen in lines 1 and 2. The *states* list is then searched for a member state that matches the current *state*. If the *state* is found in the *states* list, this indicates that the state has been visited during exploration of the current execution sequence.

The result of this search determines the value of *result* on line 3. If *result* is equal to TRUE, then the current *state* was not found in either list and considered a new state visited along the current execution sequence. If *result* equals FALSE, the state has been visited and the if block is entered to check for duplicate states. On line 5, the number of duplicate states is calculated by totaling the number of consecutive states that are identical to the current state, starting from the head of the *states* list. The value of *result* is then recomputed on line 6. If the number of duplicates is less than zero, the matching state was visited at some point prior to the last state and the value of *result* is FALSE. If the number of duplicates is greater than zero and also within the number of

allowed duplicates, *result* will be set to TRUE; otherwise it will be set to FALSE. If the framework accepts the state for further exploration, the *propertycheker* is queried and must return a TRUE value to accept the state or a FALSE to reject it. Recall that the *propertychecker's* behavior is defined by the tester and can be used to terminate select paths that violate a required sequencing of method calls or reach a user defined bound. In addition to accepting the state, this component must also evaluate whether any violations of the expected behavior have occurred. If a violation has occurred, the component will raise an exception that results in the termination of the test, a report on the nature of the violation and a printout of the sequence that caused it.

If both the *propertychecker* and the framework determine that the current path will not be terminated, lines 14 through 16 will be executed. These steps store the new state and the previous state's executioncontext information so it can be retrieved once execution backtracks to the state again. If the current path is terminated, the framework must restore the monitor to the state prior to the last transition. This task is handled by the *Backtrack* method on line 17.

When the Execute method on line 12 returns a FALSE, this indicates that all possible thread candidates have been considered for the current state. When this occurs, the framework will backtrack to the previous state, load the *ExecutionContext* for the previous state, if one exists, and move the current state the back of the *states* list. This last step is necessary to ensure correct behavior when checking for duplicates states in the *ContinueSequence* method. These actions are carried out on lines 18 through 21. If the monitor to a returned to a valid state, the *Execute* method will be called to continue

the execution sequence for the next candidate thread. In the event that the state prior to backtracking was the initial state, the *Backtrack* method will have no effect and the *ExecutionContext* stack will return a null value. If this is the case, the while loop on line 22 will exit, completing the test of the current monitor instance. If another monitor instance is available, the cycle will begin again.

3.2.1.1 Maximum Consecutive Duplicates Allowed (MCDA)

Due to the thread abstraction used when a monitor state is captured, method executions that do not change the values of state data members or the queue abstraction will result in a duplicate state. Under normal operation, the framework would consider this transition as having no effect on the synchronization behavior of the monitor resulting in termination of the current sequence and a backtrack to the previous state. There are circumstances, however, where synchronization failures can only be detected if duplicates are allowed to occur in the execution sequence.

```
   public synchronized void deposit(int value)
   {
1.      while (fullSlots == capacity){
2.         try {wait();} catch(InterruptedException e){}}
3.      buffer[in] = value;
4.      in = (in + 1) % capacity;

5.      if (fullSlots++ == 0)
6.         notify();
   }
```

Figure 3.4 The *BoundedBuffer Deposit* Method using *Notify*

One notable case is when one or more synchronized methods awaken threads using random signaling primitives instead of global signaling primitives. As an

example, consider the bounded buffer monitor from Figure 3.4 with a modified *deposit* method shown in Figure 3.4. In this variation, the *deposit* method introduces a small change that replaces the *notifyAll* call with a *notify*.

To describe the scenario, we will use the execution sequence seen in Table 3.1. Each method executed in the sequence is shown in the first column of each row entry and provides two additional pieces of information regarding the effect of the transition. The first appears in the second column and shows the state captured after each method exits the critical section. Each captured state is encased in brackets and contains two entries; the first is the value of fullSlots and the second is a set containing the thread abstraction values. Since each method contains a single *wait* primitive, the monitor's thread abstraction will include two abstract values. Remember that abstract thread values are determined by grouping all waiting or re-entry threads according to the last waiting primitive reached and assigning a value based off the group's collective state. Each group abstract value in the set will be encoded as follows: group name:abstract value. For the bounded buffer example, we will use the group names DW for the *deposit* wait group and WW for the *withdraw* wait group. Each abstract value will be encoded as either W for WAITING, E for ENTERING or B for BOTH. The second piece of information, which appears in the third column, shows an ordered list of synchronization events that occur during execution of the sequence. For each *wait* event encountered, an entry D_x or W_x will be used to represent that the x^{th} *deposit* or *withdraw* method, respectively, has been moved to the wait queue. Each *notify* call will be noted

with a N. Events in the list will be ordered from the oldest on the right most side to the most recent on the left most side.

Table 3.1 A Sample Execution Sequence

Method Called	State	Synchronization Events
W_1	[0, {WW:W}]	W_1
W_2	[0, {WW:W}]	W_1,W_2
D_1	[1, {WW:E}]	W_1,W_2, N

For this example the bounded buffer monitor will be initialized with an empty buffer and a capacity of three. The first method executed in the sequence is a *withdraw*. Since the buffer is empty, the thread cannot withdraw an item and will moved to the wait queue, resulting in the state [0, {WW:W}]. A second *withdraw* is then executed which encounters the same problem resulting in the state [0, {WW:W}]. As mentioned previously, during normal operation the framework would consider this a duplicate state having no affect on the synchronization behavior of the monitor, thereby terminating the current sequence. However, in this example we will change the framework's behavior to allow a single duplicate to be introduced into the sequence. A *deposit* is executed next, which adds a single item to the buffer and makes a single *notify* call, leading to state [1, {WW:E}]. At this point, if we look at the synchronization events we can see that there are two waiting threads, W_1 and W_2, and only one *notify* event while our state shows that we have a buffer with one item. This violates the monitor's expected behavior that all waiting threads are allowed to compete for consumption of

any new data deposited in the monitor. If the duplicate had not been allowed within the sequence, this failure would not have been detectable.

The reason duplicates are needed is because verification of the synchronization behavior in many cases must be established using completed and waiting thread information. In these cases if duplicate states are not allowed, the additional threads needed in the wait queue to verify the expected behavior would not be present and the failure will go undetected. To determine the value of the MCDA, the tester must consider what events should trigger each conditional random signaling primitive as well as the number of threads that should be affected. In the case of the deposit method of the bounded buffer, the event is the depositing of an item into an empty buffer and its expected behavior requires that all waiting *withdraw* threads be awoken. This example represents the simplest case where all threads are affected by a single event and no more than one duplicate state is needed. However, if a single event affects a variable number of threads or consecutive occurrences of the event can occur, the tester must consider both the expected behavior as well as possible monitor states that may affect the number of threads needed. As an example, consider a scenario where the bounded buffer expected behavior has been modified and now requires that one waiting *withdraw* thread be awakened each time a *deposit* occurs, assuming the wait queue is not empty. In this case the signaling event is a new item being deposited into the buffer and the number of threads affected should be at most one *withdraw* thread. To verify this expected behavior completely would require the same number of duplicate states as the capacity of the buffer. To understand why this is necessary requires consideration of the

maximum number of times this event can occur consecutively. For this example, the maximum number of consecutive deposits is equal to the capacity of the buffer. Therefore to verify that only one *withdraw* thread is awoken for each consecutive *deposit* requires *capacity* + 1 withdraw threads in the buffer, thus MCDA to be equal to the capacity of the buffer.

3.2.2 Thread Selection and Execution

The *Execute* method is responsible for simulating the different entry queue combinations that can occur due to threads reaching the monitor at different times. To simulate these combinations, the framework must maintain information regarding what thread choices are available at each state reached during the state-space exploration, as well as which choice have already been executed during a previous visit to the current state

Figure 3.5 shows the algorithm used by the *Execute* method. Lines 1 through 8 of this method are responsible for initializing the current *ExecutionContext* instance. *ExecutionContext* initialization will only occur if the current state has been reached for the first time along the current execution sequence. If the *ExecutionContext* has not been initialized, the *nextcandidate* member, seen on line 5 is set to 1. This field is used to track the next candidate thread that will be executed once the *Execute* method is called. In the next section we will out explain how the *candidatethreads* member is initialized.

.

```
Execute(){
1.   if (NOT context.isInitialized()){
2.       let context.entrycandidates be an empty list
3.       let context.nextcandidate be an integer
4.       let thread be a thread that can be initialized to execute a
                synchronized method

5.       context.nextcandidate = 1
6.       create and initialize a new thread for each executablemethods
                member and add it to the context.entrycandidates list
7.       add each globally signaled re-entry thread to the
                context.entrycandidates list
8.       add each randomly signaled thread to the context.entrycandidates
                list
  }

9.   let result = false
10.  if (context.nextcandidate <= context.entrycandidates.size()){
11.      let nextthread = context.entrycandidates.get(context.nextcandidate)
12.      nextthread.execute()
13.      context.nextcandidate = context.nextcandidate + 1
14.      result = true
     }
15.  return result
 }
```

Figure 3.5 The *Execute* Method

3.2.2.1 Thread Groups

Simulating thread competition in the entry queue requires the *Execute* method to determine what threads could be executed at each point in the execution sequence. Each entry candidate can be classified into one of three groups.

The first group consists of threads attempting to enter the critical section for the first time. The members of this group consist of all synchronized methods the user has configured the framework to execute (see Section 3.1.3) during the test of the current monitor and are stored in the *executablemethods* list. On line 6, each member of

executablemethods is used to initialize a new *thread* that is added to the *candidatethreads* list. The second group added to the *executablemethods* list on line 7, consists of all re-entry threads that have been awoken by a global primitive call. The third group added to the *executablemethods* list on line 8, consists of all waiting threads that could be selected due to one or more random signaling primitives that have occurred earlier in the execution sequence.

In order to provide a reproducible set of execution sequences across multiple executions of the framework, these thread candidates must be ordered in a consistent manner. The order of threads in group one will mimic the order in which the user has provided the methods to the framework. Since the framework manages threads in groups two and three, the ordering of these groups is dependent on the implementation of the algorithm. Our approach does not specify a specific ordering for these threads. It does, however, require that an implementation guarantee that execution of identical sequences will result in the same ordering of these threads at each state reached during an execution sequence.

Managing the threads in group three also poses an added challenge to the framework. When a random signaling primitive is normally reached, a random thread is selected from the wait queue and moved to the entry queue. However, for the framework to consider each possible thread that could be selected, this behavior cannot be allowed. To address this problem, the framework delays the selection a thread and tracks which threads were affected by the random signaling primitive. Using this approach, sequences that execute multiple random signaling primitives can result in

multiple thread selection combinations. As an example consider the following sequence of synchronization events which uses the notation described in Section 3.2.1.1:

M_1, M_2, N, M_3, N

As we can see from the sequence, at some point during execution two methods, M_1 and M_2 enter the wait queue followed by a *notify* call. As execution continues, method M_3 enters the queue followed by another *notify* call. Given this combination of waits and notifies, multiple combinations of thread selection can occur. These combinations are as follows: [M_1, M_2], [M_1, M_3], [M_2, M_1], [M_2, M_3]. Each combination listed is grouped using brackets and contains two entries, each representing the method selected by the first and second notifies respectively. Looking at the options, we can see there are three combinations that select the M_1 method. So an important question is, if the *Execute* method were to select method M_1 for execution, which thread selection combination should be used?

Since the goal of our approach is to simulate all possible thread candidates that could result due to thread competition in the entry queue, we must exercise the notifies in a manner that yields the greatest number of group three candidates as the execution sequence continues. Consider that if the second *notify* were used to awaken method M_1, the number of group three candidates at subsequent states reached along the current path would be limited to M_2. However, if the first *notify* is used, future group three candidates would contain both M_2 and M_3. To generalize this point, when the *Execute* method selects a randomly signaled thread for execution, the framework will exercise

the oldest *notify* occurring after the selected thread to ensure the greatest number of group three choices.

3.2.2.2 Thread Execution

Once execution has passed the initialization block, the *Execute* method will execute the next thread candidate. This begins by determining whether all candidate threads have been executed at the current position in the execution sequence, as seen on line 10. If an unexecuted candidate exists, it will be executed and the *Execute* method will block until the thread completes execution. If the executing thread does not return within a 30 second period of time, the framework will assume that it has reached a livelock situation and throw an exception. Once the thread returns control to the *Execute* method, the *nextcandidate* member is incremented. The method then returns either a true indicating that a candidate thread was executed or false indicating otherwise.

3.3 An Example Execution

In this section we will give a step-by-step example that shows the state-space exploration of the bounded buffer monitor seen in Figure 3.1. The monitor we will test will contain one subtle change to the *deposit* method, seen in Figure 3.6. On line 1 of this method, the while loop has been changed to an if structure that will allow any awoken thread re-entering the method after the *wait* on line 2 to continue without verifying that the buffer is not full.

```
public synchronized void deposit(int value){
1.      if (fullSlots == capacity){
2.          try {wait();} catch(InterruptedException e){}}

3.       buffer[in] = value;
4.       in = (in + 1) % capacity;

5.       if (fullSlots++ == 0)
6.          notifyAll();
   }
```

Figure 3.6 The *BoundedBuffer Deposit* Method Replacing While with If

In this example, captured states will be configured to include the *fullSlots* data member, as this is the only field that affects the monitor's synchronization behavior. The *MontiorInitialization* class will return one monitor instance that has a capacity of one. The framework will be configured to initialize the *executablemethods* list with two synchronized methods, *deposit*(1) and *withdraw*(). These methods make up thread group one for this test and will be the first two entries to appear in the *candidatethreads* list of the execution context.

Figure 3.7 shows the state diagram resulting from the state-space exploration of the bounded buffer example. Each state in the diagram displays its captured state using the notation described in Section 3.2.1.1. Transitions between states are labeled according to the action taken. Valid transition actions include *deposit* method calls that are indicated with a D_x, *withdraw* methods calls indicated by a W_x and backtracks indicated by a B. The value of x associated with a method call indicates the x^{th} execution of the same method call in the current sequence.

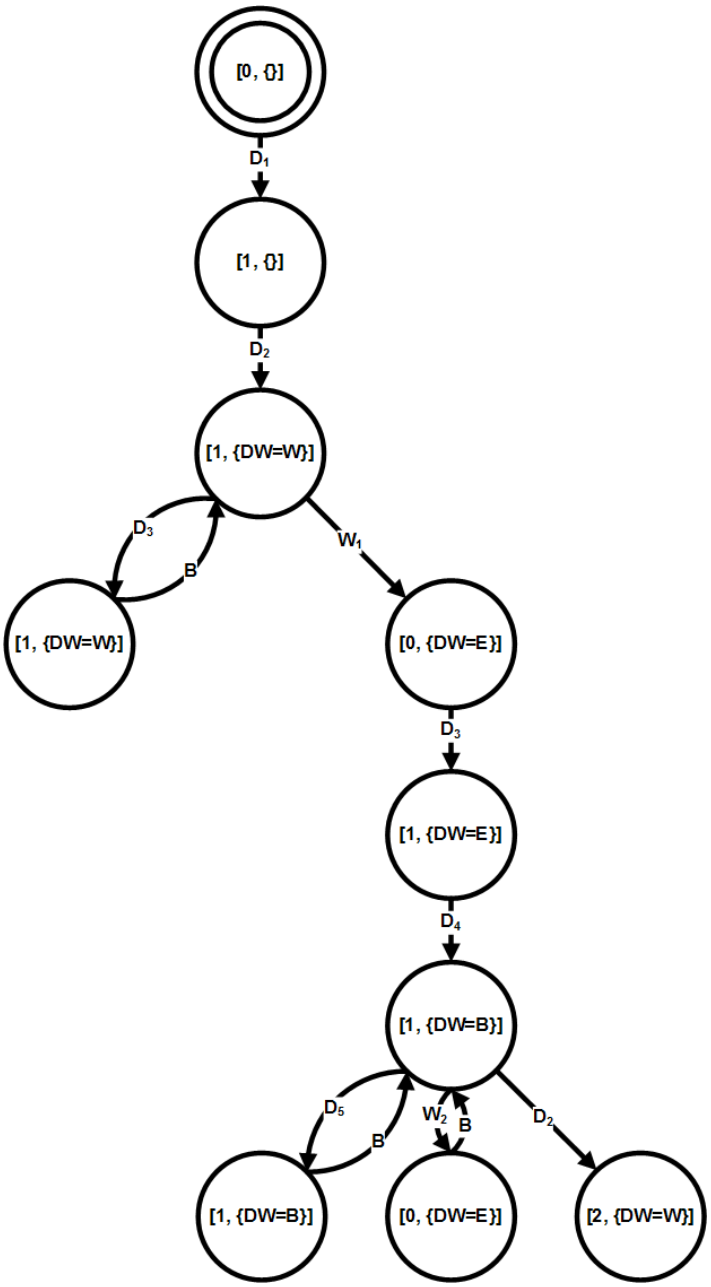

Figure 3.7 The State Diagram for the *BoundedBuffer* Example

Execution begins at the initial state [0, {}] indicating an empty buffer and no waiting threads. Once the *Execute* method is called a new *ExecutionContext* is created and a *deposit* is executed. This results in the transition labeled D_1. Once the method

exits, the monitor state [1, {}] is reached which has a buffer of 1 and no waiting threads. Since this is the first time this state has been reached, execution along the current path will continue resulting in the creation and initialization of a new *ExecutionContext*. This behavior will occur each time a state is reached that is not equivalent to another state visited along the current sequence. For the sake of brevity, each state reached that fits this description will be labeled as a new state and will result in the same set of actions.

At this point a *deposit* will be executed resulting in the transition labeled D_2. This method finds the buffer full and reaches the *wait* primitive, moving the thread to the wait queue. The result is the new state [1, {DW=W}] that shows the buffer has one item and the *deposit* wait abstract group contains only waiting threads. A third *deposit* is then executed resulting in the transition labeled D_3. This leads to the state [1, {DW=W}]. Since the *MCDA* was not overridden it has a default value of 0. Because of this, no duplicate states are allowed. Since this state is a duplicate of the last state, the framework will execute a backtrack. This action results in the transition labeled B, which restores the previous monitor state and loads its *ExecutionContext*. Since this state in the sequence has already been reached, the *nextcandidate* for this state will point to the *withdraw* method. This method is executed and results in the transition labeled W_1. The *withdraw* removes an item from the buffer, executes a *notifyAll* and exits the monitor at the new state [0, {DW=E}]. Since the thread executing D_2 has been awoken due to the *notifyAll*, it becomes a member of thread group two and will be added to the *candidatethread* list during *ExecutionContext* initialization along the current execution sequence. This will hold true until it is allowed to re-enter the monitor

or the execution sequence backtracks to the previous state. At this point, another *deposit* is executed resulting in the transition labeled D_3. The *deposit* adds a new item to the buffer, executes a *notifyAll* and exits leaving the monitor at the new state [1, {DW=E}]. A fourth *deposit* is executed resulting in the transition D_4. This method finds the buffer full and reaches the *wait* primitive, moving the thread to the wait queue. Since the monitor now has a waiting thread and a re-entry thread which are both members of the *deposit* wait abstract group, the monitor is left at the new state [1, {DW=B}]. From this state, a *deposit* is executed resulting in the transition labeled D_5. Since the buffer is full, the thread reaches the *wait* primitive and is moved to the wait queue leaving the monitor at the duplicate state [1, {DW=B}]. This duplicate will cause the current path to be terminated, causing another backtrack to the previous state. The next candidate executed is a *withdraw* which result in the transition labeled W_2. This method removes an item from the buffer and executes a *notifyAll*. Since the state reached [0, {DW=E}] has been visited along the current sequence, execution backtracks to the previous state once again. The final candidate executed is the re-entry thread executing D_2 and results in the transition labeled D_2. Because the while loop was replaced with an if construct, the method is allowed to overwrite an item in the buffer and increment the value of fullSlots to 2. This leaves the monitor at the invalid state [2, {DW=B}], which will raise a property violation exception due to having exceeded the bounds of the buffer.

CHAPTER 4

DESIGN OF THE PROTOTYPE TOOL

The design of our tool can be broken down into three major Java packages, as seen in Figure 4.1. Each package represents a different aspect of functionality that must be customized for the monitor under test before execution can begin. In the following sections we will describe the purpose of each package and the customization that must be applied.

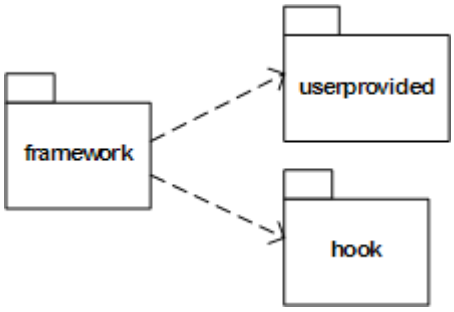

Figure 4.1 Prototype Tool Package Diagram

4.1 *UserProvided* Package

The *userprovided* package, detailed in Figure 4.2, contains 4 abstract classes and an exception class. The purpose of this package is to define the family of abstract classes that must be subclassed to implement the application specific, user provided content. The purpose of each abstract class define in this package has been outlined in the approach section should the reader need further details.

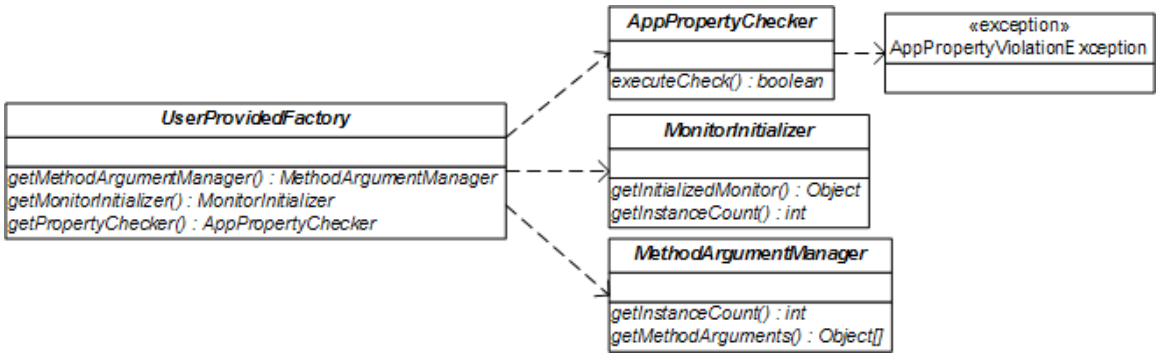

Figure 4.2 The *UserProvided* Package UML Diagram

4.1.1 The UserProvided Factory

The *UserProvidedFactory* abstract class is the first class we will discuss and provides the link between the execution framework and the user provided classes. As its name suggests, this class implements the abstract factory pattern and can be subclassed multiple time to provide different test case configurations. The class consists of 3 abstract getters each of which returns a reference to one of the user provided classes.

4.1.2 Method Assembly

The first factory method, *getMethodArgumentManager*, returns an instance subclass of the *MethodArgumentManager* abstract class. This abstract class defines the two abstract methods, *getInstanceCount* and *getMethodArguments*, which collectively are used by the framework to construct the base set of public synchronized method calls that must be executed at each new state reached during the monitor's state-space exploration. To create this set the framework must iterate through all method ids of the methods used during execution. For each method id, the *getInstanceCount* method is called, which takes the method id as an argument and returns the number of user provided argument combinations. By calling the *getMethodArguments* method, which

takes the method id and instance number as arguments, an Object array containing one combination of method arguments can be retrieved. By iterating across the full range of instance supported by the method id, all combinations are generated.

4.1.3 Monitor Instance Creation

The next factory method, *getMonitorInitializer*, returns an instance subclass of the *MonitorInitializer* abstract class. Similar to the *MethodArgumentManager*, this abstract class defines two methods, *getInstanceCount* and *getInitializedMonitor*, which return respectively the total number of monitor instance to be tested as well as the initialized monitor for a given a instance. Each monitor instance represents a new test case and a starting point for the state-space exploration.

4.1.4 Application Specific Property Checking

The last factory method, *getPropertyChecker*, returns an instance subclass of the *AppPropertyChecker* abstract class. This class' sole abstract method, *executeCheck*, provides the means by which the tester can implement an application specific properties check, evaluated at each state visited by the exploration. In the event that a property has been violated, the method must throw an *AppPropertyViolationException* exception, which will terminate execution for the current monitor instance and log details regarding the method sequence leading to the failure. In addition, this method can also be implemented to restrict illegal method sequences and terminate the current path exploration.

4.2 *Hook* Package

The *hook* package's purpose derives out of the tool's need to override the monitor scheduler and queue behavior. It consists of a class and an interface, seen in Figure 4.3, which together provide two functions.

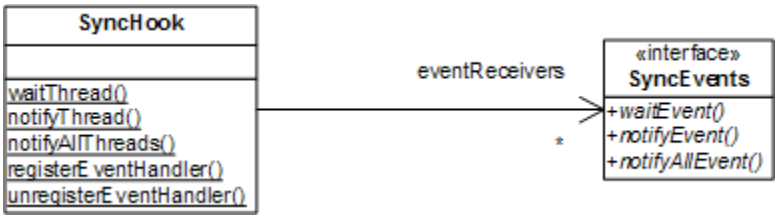

Figure 4.3 The *Hook* Package UML Diagram

4.2.1 Synchronization Event Interception

The first service provided is the interception of synchronization primitive calls. Due to the fact that true interception of these calls would require tooling at the Java implementation level, our solution opted for a language-based approach that requires the tester to replace all synchronization primitive calls with a similar method from the *SyncHook* class. All members and methods of this class are static, allowing the user to make necessary calls without creating or maintaining an instance of the class. The three interception methods provided are *waitThread*, *notifyThread* and *notifyAllThreads* which map to the Java synchronization primitives *wait*, *notify* and *notifyAll* respectively. Each method must have as an argument the synchronization object associated with the primitive call being made. This tool implementation does not yet support Java 5.0 and therefore each call will use the monitor object as its synchronization object. The *waitThread* method also requires a unique wait id that represents the wait point where

the thread exited the monitor. This additional information will be used to abstract the monitor queues each time the state is captured.

4.2.2 Synchronization Event Notification

In addition to interception, this package also provides an event service that is triggered by each incoming synchronization call. Once a call is received, all event handlers registered for the monitor from whom the call was issued, are notified. To register an event handler, the receiver must implement the *SyncEvents* interface. The *registerEventHandler* method is then called passing the monitor object that will be source of the synchronization calls and the event handler object that will be called when one or more calls are intercepted. To unregister, the receiver must call the *unregisterEventHandler* method passing the same two arguments used to register the event handler.

4.3 *Framework* Package

The *framework* package is responsible for tying together the user provided components into the test bed and controlling the overall state-space exploration. It consists of two public classes, *XMLLoader* and *ExecutionManager* seen in Figure 4.4, as well as a number of support classes, visible only at the package level.

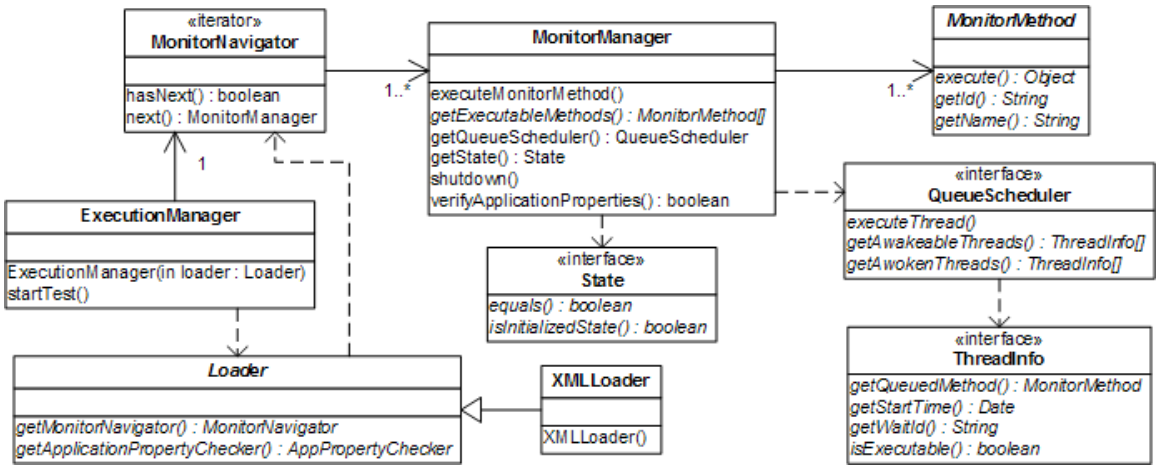

Figure 4.4 The *Framework* Package UML Diagram

4.3.1 Loading User Provided Content

One key responsibility of the framework is to locate and load all user provided content so the execution context can be initialized for testing. In order to begin this process, the tester must provide information regarding several aspects of customization. In the current implementation, the Extensible Markup Language (XML) format is used to encode this information. In order to gather the correct information and guarantee correct parsing, a Document Type Definition (DTD) file, seen in Figure 4.5, has been created which defines what information the tester must provide as well as the schema that must be used to encode the information.

The *XMLLoader* class parses the XML file created from this schema and uses its contents to initialize the user provided components of the execution framework. In order to better understand the purpose of the user provided information, we will explore each major element in the DTD file and its significance to the runtime content initialized by the loader class.

4.3.1.1 Factory Element

The factory element's sole purpose is to provide the full class name of the subclass which implements the *UserProvidedFactory* class for the current execution. This value must be in the form of a string, formatted using the full package path as well as the class name. This class, or the jar file containing this class, must be placed in a location visible to the Java runtime. This can be configured either by means of a Java command line argument, the classpath variable or inclusion in one of the default folders from which the JVM loads classes.

```
<?xml version="1.0" encoding="UTF-8"?>
<!ELEMENT userprovided (factory, monitor)>
<!ELEMENT factory (#PCDATA)>
<!ELEMENT monitor (method+)>
<!ELEMENT method (parameterTypes, (wait|notify|notifyAll)*)>
<!ATTLIST method
        name CDATA #REQUIRED
        id ID #REQUIRED>
<!ELEMENT parameterTypes (parameterType*)>
<!ELEMENT parameterType (#PCDATA)>
<!ELEMENT wait (variable*)>
<!ATTLIST wait
        id ID #REQUIRED>
<!ELEMENT variable (#PCDATA)>
<!ELEMENT notify (variable*)>
<!ATTLIST notify
        id ID #REQUIRED>
<!ELEMENT notifyAll (variable*)>
<!ATTLIST notifyAll
        id ID #REQUIRED>
```

Figure 4.5 The UserProvided DTD File

4.3.1.2 Method Element

The method elements are children of the monitor element. For each public synchronized method to be included during the test execution, there must be a corresponding method element in the XML file. Each entry must contain an id property and a name property. The id property must be a string identifier for the associated method and must be unique across all method elements listed within the XML file. This property value is the method id for the specified method and is passed to both methods of the *MethodArgumentManager* class during the method assembly phase of the loader class. The name property must be a string indicating the name of the monitor method associated with this element. This name may not be unique to the XML file if there are multiple overloaded monitor methods to be included in the test.

Because of this, the element also includes a child element that consists of parameterType elements. Each element in this collection must be a string value corresponding to a parameter type for the current method and must be listed in the collection according to its order in the method's parameter list. If the parameterType string refers to a primitive data type, then the name of the Java data type is used. If, however, the string refers to an object, the full class name must be provided, including the full package path. In either case, if the parameter is an array, the string must be suffixed with an open and closed bracket ([]). If the method has no parameters, this collection must be left empty.

The combination of the method name and parameterType collection are used by the loader class to both verify the existence of the method in the provided monitor instance as well as verify the argument array values returned by the *getMethodArguments* method.

4.3.1.3 Wait, Notify and NotifyAll Elements

In addition to a collection of parameterType elements, each method can contain zero or more wait, notify or notifyAll child elements. For each Java synchronization primitive call found within the method body, a corresponding child element is added to the xml method element. Each "synchronization" element added must define an id property, consisting of a string identifier that must be unique across all "synchronization" elements listed within the XML file.

In addition, all class variables that affect whether the synchronization primitive is executed, either as control flow or data flow variables, must be added as children to the "synchronization" element in the form of variable elements. If no variable elements are listed with a "synchronization" element, it indicates that the associated synchronization primitive will be unconditionally called each time the method is executed. During the loading phase, the loader class compiles a master collection of variable elements for all wait, notify and notifyAll elements. This collection should provide a complete list of the class variables affecting the synchronization behavior of the monitor and as such are captured in addition to the queue abstraction each time the monitor is requested.

4.3.2 Test Execution

To begin execution, the user must create an *ExecutionManager* object by calling its constructor with an object that is a subclass of the abstract *Loader* class. Currently, the *XMLLoader* is the only option available due to the fact that user provided data has only been persisted in the XML format. Future versions of the framework, however, could offer alternative loader classes that gather initialization data from databases, flat files, user interfaces, or web services to name a few. The purpose of the *Loader* and its subclasses are to separate the format in which the user data is persisted from the complex internal objects constructed using this data. This concept is similar to that of the Builder pattern.

Once the loader has completed, it provides the framework with two objects that encapsulate the user provided logic used during execution. The first is the property checker object, called each time a new state is reach during exploration, and the second a *MonitorNaviagor* object. The *MonitorNaviagor* is a simple iterator whose purpose is to return *MonitorManager* objects and controls the outer loop of the execution algorithm. These *MonitorManager* objects are complex and encapsulate all the resultant user defined components generated from the XML file. As seen in the UML diagram in Figure 4.4, they define methods that can be used to capture the monitor state, execute new threads, or manage the monitor queues. They also maintain a collection of *MonitorMethod* objects, each of which represents a combination of a method to be tested and a set of user provided arguments.

Once the framework has been initialized, the user begins the run by calling the *startTest* method of the *ExecutionManager* object, which encapsulates the execution algorithm (section 3.2.1). In addition, the *ExecutionManager* supplies implementations to verify valid states, new states, duplicate states and productivity during execution as well as the rollback method called at each point where a path has been terminated. A set of inner classes implements a state pattern that is responsible for maintaining the execution context for each state visited and implementing the *execute* functionality.

The only significant difference between the conceptual algorithmic approach and the tool's implementation arose in the backtrack functionality. To backtrack the monitor, it would be necessary to not only restore the state of any class or global variables that have changed, but also to restore the previous state of the monitor queues as well as the execution frames. Because the language does not natively support this functionality, an alternative approach was used. At each point requiring a backtrack, the current *MonitorManager* instance was shutdown and a new *MonitorManager* instance was created. To reconstruct the monitor state, the sequence of execution contexts that provide the last method executed at each state along the last path, are executed. The results of each test are outputted to the console.

CHAPTER 5

CASE STUDIES

5.1 Experimental Design

A mutation-based approach was used to evaluate the detection ability of our prototype tool [16][31]. To conduct each case study, a collection of mutants was created for each correctly coded monitor used. Each mutant created contains a single change to the source code of the original monitor and represents a programming mistake that can occur during the implementation of a Java monitor. In each case study, two general groups of mutants were created.

The first group of mutants represents typical errors made by programmers when implementing the logic used in a monitor. These mutants were created using a publicly available, Java based mutation tool called μJava [34][39]. This tool creates mutants that change the class level or method level operators used in the source of a Java class. Class level mutants introduce syntactic faults to the object oriented operators used in the source such as changing the access modifier or static modifier associated with a class, method or variable. Because we are only interested in testing the synchronization behavior of the monitor, class level mutants were not used. Method level mutants introduce faults to common operators used to implement the body of Java methods. For the version of μJava we used, the operators that were mutated include arithmetic

operators, relational operators, conditional operators, shift operators, logical operators and assignment operators.

The second group of mutants introduces a change that affects the synchronization primitives or control flow structures used to implement the synchronization behavior of the Java monitor. Each monitor created for this group reflects one of the following changes:

- Replacement of a *while* loop which encases a *wait* primitive with an *if* statement.
- Replacement of a *notifyAll* primitive with a *notify* primitive.
- Removal of a *wait*, *notify* or *notifyAll* synchronization primitive.

Each of the mutants created was tested using our prototype tool and executed using the Sun Microsystems Java Runtime Environment 1.5. All results were generated on a computer running Windows XP with a 2GHz processor and 1 GB of ram.

5.2 Monitor Tests

In the following section we will describe the monitors and framework configurations used in each case study. The monitors used for each case study were taken from [33]. Each subsection starts by describing the purpose of the monitor and its expected behavior. This will be followed by an explanation of how the user provided components initialized the framework for each mutant tested. Recall that the tester is responsible for providing an XML file that defines the methods to be included in each test and the data members used in the state representation as well as implementations of the *MonitorInitialization*, *MethodArguments* and *PropertyChecker* classes. Collectively these implementations determine the number of different initialized monitor instances

tested, the argument combinations for each method included in the test and the verification and state acceptance properties evaluated at each state reached during the state-space exploration. The original monitor source and configuration XML file used in each case study can be found in Appendixes A through E.

5.2.1 BoundedBuffer Monitor

The *BoundedBuffer* monitor implements a solution to the producer/consumer problem. This monitor consists of two synchronized methods, *deposit* and *withdraw*, which allow items to be added and removed from a fixed size buffer. Each time a *deposit* method is executed, a data item will be added to the buffer as long as the buffer is not full. After the data is added to the buffer, any *withdraw* threads in the wait queue must be woken up so they can compete to consume the new data. If the buffer is full, the calling thread must be moved to the wait queue until a data item is removed from the buffer. When the *withdraw* method is called a data item will be removed from the buffer as long as the buffer is not empty. If an item is removed from the buffer, any *deposit* threads in the wait queue must be woken up so they can compete to deposit their data in the buffer. If the buffer is empty, the calling thread must be moved to the wait queue until a data item is added to the buffer.

In this case study, 113 mutants were tested of which 101 were members of the first mutant group and 8 were members of the second mutant group. For each test executed, a single instance of the mutant was used that was initialized with an empty buffer and capacity of five. The member *fullSlots* is an integer value that represents the number of used slots in the monitor's buffer. Since it is the variable that affects the

synchronization behavior of the monitor it was defined as the data member used in the state representation. The *get*(1) and *put*() methods were defined to be the two methods executed during the state-space exploration. Though the *get* method passes an integer argument each time it is called, this value does not have an affect on the monitor's synchronization behavior and any integer value could have been chosen. The *PropertyChecker* was implemented to accept all states reached. The properties used to verify each state reached are based on the exercised synchronization behavior of the monitor during the execution of the current sequence. To this end, each variable upon which the synchronization behavior is dependent has been calculated at each state using the following formulas:

- *fullSlots* = # of initial slots used + # completed *deposits* - # completed *withdraws*
- *capacity* = the *capacity* value used to initialize the monitor instance

Based on these values, the properties verified during each test are as follows:

- 0 <= fullSlots <= capacity
- 0 < fullSlots < capacity $\rightarrow$ the wait queue should be empty
- fullSlots == 0 $\rightarrow$ the wait queue should be contain no deposit threads
- fullSlots == capacity $\rightarrow$ the wait queue should contain no withdraw threads

5.2.2 ReaderWriterSafe Monitor

The *ReaderWriterSafe* monitor implements a solution to the readers/writers problem. In this problem there exists a shared variable that multiple threads are attempting to either to either read or write. Before each thread accesses the shared variable, it is required to acquire a lock that guards the shared variable. A reader may

acquire the lock only if no writer has currently acquired the lock. This behavior allows multiple readers to access the shared variable at the same time. A writer may acquire the only if no other reader or writer has currently acquired the lock. This guarantees that only one writer will have mutually exclusive access to the shared variable at a time. Once a thread has finished accessing the variable it must release the lock and wake up all other waiting threads if no other thread has the lock. The *ReaderWriterSafe* monitor allows threads to acquire and release locks using one of four synchronized methods. If a thread wants to acquire the lock for read access it must call the *acquireRead* method. Once the thread has completed it must then call the *releaseRead* method. Likewise, the pair of methods *acquireWrite* and *releaseWrite* must be called to acquire and release a lock for write access.

For this case study, 28 mutants were created of which 20 were members of the first mutant group and 8 were members of the second mutant group. A single mutant instance was used for each test executed that was initialized using the empty constructor. In each test the framework was configured to include all four synchronized methods of the monitor. Two data members in the *ReaderWriterSafe* monitor have an effect on the synchronization behavior of the monitor. The *readers* member is an integer that keeps a count of the number of readers that have current acquired a read lock while the *writers* member is a boolean that indicates whether a write lock has been acquired. Both of these data members were used in the state abstraction. As described in the section, the variables and properties used to verify each state reached are based on the exercised synchronization behavior of the monitor and were defined as follows:

Variables

- *writers* = # of completed *acquireRead* threads - # of completed *releaseRead* threads
- *readers* = # of completed *acquireWrite* threads - # of completed *releaseWrite* threads

Properties

- *writers* > 0 $\rightarrow$ *readers* == 0
- *writers* == 0 $\rightarrow$ the wait queue should contain no waiting *acquireRead* threads
- *writers* == 0 AND *readers* == 0 $\rightarrow$ wait queue should be empty
- *readers* > 0 $\rightarrow$ *writers* == 0
- *writers* > 0 $\rightarrow$ *writers* == 1

In addition, the *PropertyChecker* was configured to accept states only if 0 <= *readers* <= 1 AND 0 <= *writers* <= 1 AND (# of completed *acquireRead* <= 2 AND # of completed *acquireWrite* threads <= 2. The first two conditions guarantees the implicit lock unlock protocol for both readers and writers as well as bounds the number of locks at any given time to 1. The last two condition provides an upper bound of 2 on the number of total number of locks that can be acquired in any execution sequence.

5.2.3 FairBridge Monitor

The *FairBridge* monitor implements a solution that is designed to prevent cars coming in opposite direction from colliding on a one-lane bridge. To guarantee a

measure of fairness, a turn-based system is used. Cars from each side of the bridge are allowed to cross the bridge only if it is their turn or no other cars are waiting on the other side. In either case, a car must wait until the bridge is empty of cars coming in the opposite direction before it can cross. Because of this, the last car to cross the bridge must signal to any waiting cars from the opposite direction that they may now cross. Once a car from one side crosses the bridge, it becomes the turn of the other side. The *FairBridge* monitor allows cars to enter and exit the bridge using four synchronized methods. Each side of the bridge is represented by the colors blue and red. Cars coming from the red side enter the bridge by calling the *redEnter* method and exit the bridge by calling the *redExit* method. This same behavior is implemented for the blue side with the methods *blueEnter* and *blueExit*.

For this case study, 61 mutants were created of which 53 were members of the first mutant group and 8 were members of the second mutant group. Each test executed, used a single mutant instance that was created by calling the empty monitor constructor. The synchronized methods executed during each test were *redEnter*(), *redExit*(), *blueEnter*() and *blueExit*(). The values of *nblue*, *nred*, *waitred*, *waitblue* and *blueturn* all have an affect on the synchronization behavior of the monitor. Members *nblue* and *nred* are integer values that represent the number of cars currently on the bridge from the blue side or the red side. The *waitred* and *waitblue* members are integer values that represent the number of cars from the red side or blue side that are waiting to enter the bridge and the *blueturn* member is a boolean value that tracks whether it is blue's turn or red's turn. Each of these values is used as data members in the state representation.

Similar to the previous case studies, the variables and properties used to verify each state reached are based on the exercised synchronization behavior of the monitor as follows:

Variables

- *blueCarsOnBridge* = # of completed *blueEnter* threads - # of completed *blueExit* threads
- *redCarsOnBridge* = # of completed *redEnter* threads - # of completed *redExit* threads
- *bluesTurn* = (# of completed *blueExits* == 0 AND # of completed *redExits* == 0) OR (# of completed *redExits* != 0 AND the last *blueExit* is older than the last *redExit*)
- *blueCarsWaiting* = # of *blueEnter* threads in the entry and wait queues
- *redCarsWaiting* = # of *redEnter* threads in the entry and wait queues

Properties

- *blueCarsOnBridge* > 0 → *redCarsOnBridge* == 0
- *blueCarsWaiting* > 0 AND *bluesTurn* → no red cars should be on the bridge that entered after it became blue's turn AND after the longest waiting blue car arrived
- *blueCarsOnBridge* == 0 AND (*blueCarsWaiting* == 0 OR NOT *bluesTurn*) → the wait queue should not contain *redEnter* threads
- *blueCarsOnBridge* == 0 AND *redCarsOnBridge* ==0 AND *bluesTurn* → the wait queue should not contain *blueEnter* threads

- *blueCarsOnBridge* == 0 AND *redCarsOnBridge* ==0 AND NOT *bluesTurn* → the wait queue should not contain *redEnter* threads
- *redCarsOnBridge* > 0 → *blueCarsOnBridge* == 0
- *redCarsWaiting* > 0 AND NOT *bluesTurn* → no blue cars should be on the bridge that entered after it became red's turn AND after the longest waiting red car arrived
- *redCarsOnBridge* == 0 AND (*redCarsWaiting* == 0 OR *bluesTurn*) → the wait queue should not contain *blueEnter* threads

Acceptance of states reached during the state-space exploration was determined based off the following conditions: 0 <= *blueCarsOnBridge* <= 1 AND 0 <= *redCarsOnBridge* <= 1 AND *blueCarsWaiting* <= 2 AND *redCarsWaiting* <= 2 AND # of completed *blueExits* <= 2 AND # of completed *redExits* <= 2. The first two conditions guarantee that the correct order of enters and exits are followed according to the expected bridge protocol as well as bounds the number of cars on the bridge to no more than 1. The next two conditions ensure that no sequence can have more than two cars waiting on either side of the bridge. The last two conditions bound the number the number of times that each side can cross the bridge in any execution sequence to a maximum of 2.

5.2.4 FairAllocator Monitor

The *FairAllocator* monitor is implemented to manage the allocation of a pool of balls in a first-come first-serve manner. Conceptually, each thread that interacts with the monitor represents a customer. Customers wanting to request one or more balls must

form a line according to the order in which they arrived. If there are enough balls to service the request of the first customer in line, the requested balls will be allocated from the ball pool to the customer and the next customer in line can submit their request. If the first customer in line requests more balls than are currently available in the ball pool, they and all other customers in the back of the line must wait until enough balls are returned to fill the request. Once a customer has finished using the balls they checked-out, they must return them to the ball pool. The *FairAllocator* monitor allows threads to request and return balls on behalf of a customer using the synchronized methods *get* and *put*.

For this case study, 60 mutants were created of which 54 were members of the first mutant group and 6 were members of the second mutant group. For each mutant tested, a single monitor instance was created with an initial ball pool of size 2. The *get* synchronized method defined by the *FairAllocator* monitor passes an integer argument that represents the number of balls requested by the thread. Since this value affects the synchronization behavior of the monitor, two argument combinations were defined for the *get* method, namely *get*(1) and *get*(2). In addition, two argument combinations were defined for the *put* method resulting in *put*(1) and *put*(2) also being executed by the framework. The data members captured as part of the state representation for this monitor include *available* and *next* and were chosen because they affect the synchronization behavior of the monitor. The *available* member is an integer value that tracks the number of balls currently in the ball pool while the *next* member is an integer that determines who is the next thread at the front of the line. Similar to the previous

case studies, the variables and properties used to verify each state reached are based on the exercised synchronization behavior of the monitor as follows:

Variables

- *ballsCheckedOut* = the sum of balls checked out by completed *get* threads - the sum of balls returned by completed *put* threads
- *totalBalls* = the value used to initialize the monitor's available ball count (i.e., 2)
- *availableBalls* = *totalBalls* - *ballsCheckedOut*

Properties

- *ballsCheckedOut* <= *totalBalls*
- # of waiting *get* threads > 0 AND last executed thread == longest waiting *get* thread $\rightarrow$ # of balls requested by longest waiting *get* thread > *availableBalls*
- no completed *get* threads began execution after the longest waiting *get* thread
- last executed thread completed $\rightarrow$ the wait queue should be empty

The *PropertyChecker* implementation for each test was configured to accept states only if *ballsCheckedOut* >= 0 AND (# of completed *get* threads) <= *totalBalls* * 2. The first condition is required to guarantee the correct sequencing of *get* and *put* calls that collectively must only return up to as many balls as have been checked out. The last condition bounds the number of threads that can execute a *get* to 2 times the number of total balls in the pool.

5.2.5 BoundedOvertakingAllocator Monitor

The *BoundedOvertakingAllocator* monitor was designed to manage the allocation of balls in a similar manner as the *FairAllocator* described in Section 5.2.4. The goal of the FairAllocator monitor is to provide a fair allocation protocol that does not favor smaller ball requests that can be filled immediately but rather to service requests according to the order in with the requests are made. This type of protocol can create a situation where the first customer in line requests a large number of balls, forcing other customers who may only want a small number of balls to wait for a long before their request can be filled. To address this concern, the *BoundedOvertakingAllocator* monitor implements a compromise. If the first customer in line requests more balls than are currently available, the monitor will allow up to a fixed number of customers whose order can be filled immediately to overtake all other waiting customers and have their requests serviced. To ensure fairness as the line moves forward, no customer may be overtaken by more than the fixed upper bound of other customers during the time they are waiting in the line.

The framework in this case study was configured to supply a single monitor instance with a ball pool of size two and an overtaking upper bound of 1. The synchronized methods executed were identical to those used in the *FairAllocator* configuration. Similarly, the *available* and *next* monitor members were still used in the state representation. However, since the synchronization behavior of the monitor is also dependent on the *overtaken* member, it has been added to the state representation as well. The variables and properties defined for the *BoundedOvertakingAllocator* monitor

are also similar to those defined for the *FairAllocator* monitor. The only change made was to the property "no completed *get* threads began execution after the longest waiting *get* thread" which was changed to "number of completed *get* threads that began execution after the longest waiting *get* thread <= *overtakingLimit*", where *overtakingLimit* is equal to the limit argument passed to the monitor's constructor. This change reflects the new overtaking policy and verifies that each *get* thread at the front of the line has not been overtaken more times than the allowed upper bound. The *PropertyChecker's* implemented state acceptance for this monitor is identical to that of the *FairAllocator* monitor.

In this case study no mutants were created and tested. This is due to the fact that when the original monitor was executed against our tool, a property violation occurred when the execution sequence seen in Table 5.1 was reached. This sequence begins with the thread T_1 executing *get*(1) that results in the allocation of one of the two balls in the monitor ball pool. When thread T_2 executes *get*(2), there are not enough balls to fill the request so it is moved to the wait queue. Thread T_3 then executes *get*(1) and overtakes T_2 since there are enough balls to fill its request and no other threads have overtaken T_2. Since the ball pool is now empty, thread T_4 is moved to the wait queue when it executes *get*(1). The next two threads execute a *put*(1), which returns two balls back to the ball pool. At this point there are two awoken threads, T_2 and T_4, which are competing for the returned balls. Thread T_2 is at the head of the line and has been overtaken once by T_3. Since the monitor has been configured to allow each waiting thread to be overtaken

only once, it should not be possible for T_4 to have its request filled before T_2, however, this action was allowed by the monitor.

Table 5.1 The *BoundedOvertakingAllocator* Property Violation Sequence

Thread	Method Executed
T_1	*get*(1)
T_2	*get*(2)
T_3	*get*(1)
T_4	*get*(1)
T_5	*put*(1)
T_6	*put*(1)
T_4	*get*(1)

5.3 Results

Table 5.2 Original Monitor Test Results

Monitor	# of States Visited	Paths Explored	Transitions	Execution Time
BoundedBuffer	24	49	72	0.19 Seconds
ReaderWriterSafe	72	120	147	0.28 Seconds
FairBridge	354	385	470	0.96 Seconds
FairAllocator	92	172	214	0.39 Seconds
BoundedOvertakingAllocator	8	4	12	0.4 Seconds

Table 5.3 Mutant Test Results

Monitor	# of Mutants	# of Mutants Detected	Avg # of States/Paths/Transes	Avg/Total Exe Time
BoundedBuffer	109	75	12/17/28	0.12/13.39 Seconds
ReaderWriterSafe	28	25	20/26/35	0.04/11.17 Seconds
FairBridge	61	57	90/81/104	0.33/19.86 Seconds
FairAllocator	60	54	14/21/29	0.12/7.09 Seconds

Table 5.2 and Table 5.3 show the results collected from testing each of the original monitors and mutants in each case study. In each case study a number of the mutants tested did not raise a property violation during the execution of their test. In the case of the *BoundedBuffer*, 18 of the undetected mutants contained a mutation in the source that affected the value of either the *in*, *out* or *value* variables. The *in* and *out* variables are used to manage the storage and retrieval of the data items being deposited and withdrawn from the buffer and do not affect the synchronization behavior of the monitor. The *value* variable is used to temporarily store the value of a data item being deposited and withdrawn from the buffer and likewise does not affect the synchronization of the buffer. Since the properties being verified are designed specifically to detect synchronization failures it is expected that these mutants were not detected. The remaining undetected mutants in each of the case studies were found to implement a synchronization behavior that is equivalent to their original monitor and thus would violate the properties being verified.

When we compared the results of the BoundedBuffer and ReaderWriterSafe case studies to the results generated by the MonitorExplorer [29] for these monitors the following observations were made. The MonitorExplorer results for the original BoundedBuffer explored 33 states and 47 transitions and took a total of 2.2 seconds and the results for the original ReaderWriterSafe explored 75 states and 106 transitions and took 3.65 seconds. In both cases our tool explored less states, however, executed a larger number of transitions. Due to the fact that both tests did not use equivalent hardware, our approach does not use data member abstraction in our state representation and that no information was provided regarding the number of paths explored no conclusions can be drawn from these observations.

CHAPTER 6

CONCLUSION AND FUTURE WORK

In this thesis we have presented a dynamic framework for testing the synchronization behavior of Java monitors. This framework uses a state-space exploration based approach that is driven by dynamically created sequences of synchronized method execution calls. Each state visited in the state-space represents the monitor under test with no thread executing in its critical section and all enabled transitions for a state simulate possible thread interactions that could occur given the current monitor state. By creating new threads on the fly as needed, each execution sequence simulates a possible combination of thread interaction with the monitor, which in turn exercises a combination of the monitor's synchronization behavior. This simplifies testing of a Java monitor by relieving the tester of having to design test cases that exercise the synchronization behavior and also consider the non-deterministic arrival of threads to the entry queue. Because each synchronized method executed may affect the state of monitor data members as well as the monitor queues, verification that each state reached is consistent with the expected synchronization behavior of the monitor must be ensured using application-specific properties provided by the tester. If these properties are not accurate or do not sufficiently cover the expected synchronization behavior of the monitor, execution of a test may result in a false positives that will be difficult for the tester to detect.

The framework makes several assumptions about the monitor under test that currently limit the type of monitors that can be tested. Each method that is to be executed during a monitor test must be a synchronized method. Condition synchronization implemented in each method must currently use only a combination of *wait*, *notify* and *notifyAll* synchronization primitives. No support is current available for synchronized blocks, timed waits or synchronization constructs added as of Java 5.0. As a result, we would propose extending the framework to add support for these features, as this will allow the approach to test industry level monitor implementations. Data member captured at each state reached must be of a primitive data type. This reduces the level of user configuration, however, limits the state-space explored in cases where the synchronization behavior of the monitor is dependent on the value of a complex data type or on an object instance. We would propose an extension to the data member representation that would allow the user the option of data abstraction in situations where data members are not of a primitive data type. This additionally could be used to represent data values that may iterate over a large range of values yet only specific values are significant to evaluating synchronization behavior of the monitor given the current configuration. Currently, the approach handles these situations by either allowing the exploration to exercise all possible values for these types of data members or terminate a path based on user defined bound. By terminating a path based on a fixed upper or lower bound, the approach does not have the flexibility to explore additional states along the current path, which could be reachable using data abstraction.

The framework is currently designed to test a single monitor instance in isolation. Though a monitor is designed to be accessed by multiple threads at the same time, due to the mutual exclusion provided by the synchronized methods, only one thread will be allowed to execute within the critical section at any given time. As a result, the framework currently assumes that all thread interaction with encapsulated data members within the critical section (i.e., data primitives, complex data types or object references) are thread safe. Since deterministic execution is implemented by a sequential execution of threads within the critical section, there is no benefit in executing a test on a multi-processor hardware.

Though the framework is designed to interact with a single monitor instance, it could be extended to test multiple instance of a monitor in parallel. This extension could be used to verify the synchronization behavior of the monitor when simulating its execution on a multi-processor or distributed system where multiple instances may interact with common resources. Such an extension would require the exploration driver to provide deterministic execution of multiple instances of the monitor that simulate different interleavings of thread interaction across each monitor instance. It would still be assumed however that each data member that could be accessed within the critical section provides a thread safe implementation. It should be noted here that since the approach focuses on detecting failures in the synchronization behavior of the monitor, detecting faults due to thread interaction with shared data members between the monitor instances would be limited to those shared data members that affect the synchronization

behavior of the monitor. In the event that a shared data member is not thread safe, the framework may return false positives or false negatives due to race conditions.

This framework has demonstrated success in detecting synchronization failures in each of the case studies described in Section 5. The only mutants that were not detected either did not affect the synchronization behavior of the monitor or implemented an equivalent synchronization behavior to that of the original monitor. This level of detection was the same as that achieved by the MonitorExplorer [29] when testing the *BoundedBuffer* and *ReaderWriterSafe* monitors in the first two case studies.

To discuss the potential costs of our approach against that of the MonitorExplorer, we must consider the effect of the difference between the two. The key differences between these two approaches are in the way states and transitions are defined and the rules used for introducing new threads. In the MonitorExplorer, transitions are defined at a level of granularity that is consistent with visible operations of the monitor. Each executed transition therefore represents either a thread entering the critical section, executing a *notify*/*notifyAll* or a *wait*, exiting the critical section or the introduction of a new thread into the entry queue. In our approach, transitions represent a thread entering the critical section, executing one or more operations within the critical section and eventually exiting the critical section, either due to completion of the method or having reached a *wait* operation. As a result of these differences, the states visited in our approach represent a monitor at each time when no thread is executing within the critical section while the MonitorExplorer may visit several states during the execution a thread within the critical section. This reduction in the number of states

visited during the execution of a thread in the critical section is offset by the differences in state representation and enabled transitions that have a greater impact on the size of the state-space that can be visited for each monitor.

Let us first consider the impact of the state representation used. In both approaches data values defined by the user are captured at each state are and are assumed to represent the data members that affect the synchronization behavior of the monitor. The MonitorExplorer uses abstraction to represent the data values at each state while our approach uses concrete values. As we mentioned previously, data abstraction can offer a greater degree of flexibility during exploration of a path, which can allow an approach to explore additional states that cannot be reached by explicit termination of a path due to a data value having reached an upper or lower bound. A limitation in using data abstraction is that based on the implementation of the monitor, there may be cases in which several tests, each requiring a different initial configuration of the monitor, are required to provide sufficient coverage. This can result when the data abstraction chosen for a specific test configuration does not sufficiently explore the synchronization behavior of the monitor. Therefore if the range of possible values each data member will exercise is kept reasonable by the test configuration, capturing the concrete values of the monitors can simplify the configuration process. In addition, the use of data abstraction requires an understanding of the implementation to determine the abstract values that will be used for each data member and tightly couples the test configuration to the monitor's implementation rather than its specification. Both approaches also use thread abstraction to represent the state of select threads interacting with the monitor. In

the MonitorExplorer the thread abstraction consists of abstract values assigned for three aspects of the monitor. The first is an abstract value representing the thread currently executing in the critical section. The second is an abstract value representing the next thread in the entry queue that will be allowed into the critical section once it is empty. The last is a type vector consisting of abstract values representing threads in the wait queue. Each entry in the type vector maps to a executable synchronized method in the test that contains at least one *wait* primitive and the abstract value assigned for each entry is determined by whether there is at least one thread in the wait queue which has reach one of the method's *wait* primitives. In our approach the thread abstraction represents the state of threads in both the entry queue and wait queue where each abstract value assigned is based collective state of threads according to the last *wait* primitive reached. Representing the state of threads based off the last *wait* point reached provides a finer level of granularity than thread states defined at the method level, however, can quickly result in a much larger state-space which must be bound by a test configuration as it can result in state explosion. As an example, let's consider that the number of possible combinations for the thread abstraction used in each approach. For the MonitorExplorer the combinations equal $(M + 1)^2 * 2^{Mw}$ where M equals the number of synchronized methods executed in each test and M_w equals the number of synchronized methods executed in each test that contain at least one *wait* primitive. In our approach the combinations equal 4^W where W equals the number of *wait* primitives across all synchronized methods executed in each test. It should be noted that these equations represent an upper bound on the number of states that can be visited based

solely on the thread abstraction used in each state representation, however, not all of these combinations will be reachable during a test of a monitor. If we consider a simple case where each synchronized method executed during a test contains a single *wait* primitive, the number of combinations for our approach will exceeded those of the MonitorExplorer for a test with six or more methods. In situations where synchronized methods contain a greater number of *wait* primitives, the number of combinations for our approach will exceed that of the MonitorExplorer with a smaller number of synchronized methods used in a test. This difference represents a significant cost that can be incurred for more complex monitors, however, the potential benefit is that the thread representation used is more sensitive to the implemented synchronization behavior of the monitor and therefore will reflect a finer state of the monitor.

The enabled transitions in both approaches are designed to simulate different possible thread interactions with the monitor during the state-space exploration. At each state where no thread is executing within the critical section, the enabled transitions in both approaches contain transitions to represent the entry of any threads that have been awakened along the current path into the critical section or in the event that no awakened threads are in the entry queue, transitions representing the introduction of a new thread for each synchronized method that will compete for entry into the critical section. The difference between the two approaches is the rules used to determine when transitions will also be included to represent the possibility of new threads being allowed to compete with awaken threads for entry into the critical section. In the MonitorExplorer the decision to introduce new threads to compete with awakened

threads is made at each state reached during the exploration where a *notify*/*notifyAll* is the next visible operation to be executed by the thread in the critical section. In this situation the enabled transitions will include transitions for each awakened thread as well as transitions to introduce competing threads for each possible synchronized method that can be executed during the test. This results in the possibility of a single new thread being allowed to enter into the critical section before an awakened thread is allowed to re-enter the critical section. This differs from our approach that allows a variable number of possible competing new threads to be allowed into the critical section before an awakened new thread is allowed to re-enter. This results in a larger number of possible paths that can be explored for each awakened thread, which will increase the cost for each test as the complexity of the conditional synchronization increases.

Given the differences in the way the MonitorExplorer and our approach define states and transitions, it is impossible to perform a one-to-one comparison between the two approaches. Each difference that we have described represents costs that can be incurred during the state-space exploration of the monitor with the potential benefit of added coverage in specific situations. The cost to benefit ratio will largely be dependent on the complexity of the monitor under test as well as the test configuration defined by the user. Due to differences in testing hardware, the type and number of mutants tested in the two overlapping case studies and the type of data recorded, there is a lack of data to make any further analytical observations between these two approaches. Therefore we would like to conduct additional case studies to further evaluate the coverage and

performance of this approach against monitors of increasing levels of complexity both in terms of methods executed during a test but also in terms of the condition synchronization implemented by each method. As part of this effort we would like to compare the fault detection and performance costs of our approach with that of the MonitorExplorer to determine whether our potential added coverage justifies the inherent added costs.

APPENDIX A

BOUNDEDBUFFER CASE STUDY

Correct BoundedBuffer Source Code

```
import edu.uta.monitortester.hook.SyncHook;

public class BoundedBuffer {

   private int fullSlots = 0;
   private int capacity = 0;
   private int[] buffer = null;
   private int in = 0, out = 0;

   public BoundedBuffer(int bufferCapacity)
   {
      capacity = bufferCapacity;
      buffer = new int[capacity];
   }

   public synchronized void deposit(int value)
   {
      while(fullSlots == capacity)
      {
         try
         {
            SyncHook.waitThread(this, "deposit_wait_1");
         }
         catch(InterruptedException ex) {}
      }

      buffer[in] = value;
      in = (in + 1) % capacity;

      if (fullSlots++ == 0)
         SyncHook.notifyAllThreads(this, "deposit_notify_1");
   }

   public synchronized int withdraw()
   {
      int value = 0;
      while(fullSlots == 0)
      {
         try
         {
            SyncHook.waitThread(this, "withdraw_wait_1");
         }
         catch(InterruptedException ex) {}
```

```
        }
        value = buffer[out];
        out = (out + 1) % capacity;

        if (fullSlots-- == capacity)
            SyncHook.notifyAllThreads(this, "withdraw_notify_1");

        return value;
    }

}
```

BoundedBuffer XML Configuration File

```
<?xml version="1.0"?>
<!DOCTYPE userprovided SYSTEM "UserProvided-schema.dtd">

<userprovided>

<factory>test.boundedbuffer.userprovided.BoundedBufferUserProvidedFactory</factor
y>
   <monitor>
      <method name="deposit" id="deposit">
         <parameterTypes>
            <parameterType>int</parameterType>
         </parameterTypes>
         <wait id="deposit_wait_1">
            <variable>fullSlots</variable>
         </wait>
         <notifyAll id="deposit_notify_1">
            <variable>fullSlots</variable>
         </notifyAll>
      </method>
      <method name="withdraw" id="withdraw">
         <parameterTypes>
         </parameterTypes>
         <wait id="withdraw_wait_1">
            <variable>fullSlots</variable>
         </wait>
         <notifyAll id="withdraw_notify_1">
            <variable>fullSlots</variable>
         </notifyAll>
      </method>
   </monitor>
</userprovided>
```

APPENDIX B

READERWRITERSAFE CASE STUDY

Correct ReaderWriterSafe Source Code

```
import edu.uta.monitortester.hook.SyncHook;

public class ReaderWriterSafe
{
   private int readers =0;
   private boolean writing = false;

   public synchronized void acquireRead() throws InterruptedException
   {
      while (writing)
         SyncHook.waitThread(this, "acquireRead_wait_1");
      ++readers;
   }

   public synchronized void releaseRead()
   {
      --readers;
      if(readers==0)
         SyncHook.notifyAllThreads(this, "releaseRead_notify_1");
   }

   public synchronized void acquireWrite() throws InterruptedException
   {
      while (readers>0 || writing)
         SyncHook.waitThread(this, "acquireWrite_wait_1");
      writing = true;
   }

   public synchronized void releaseWrite()
   {
      writing = false;
      SyncHook.notifyAllThreads(this, "releaseWrite_notify_1");
   }
}
```

ReaderWriterSafe XML Configuration File

```
<?xml version="1.0"?>
<!DOCTYPE userprovided SYSTEM "UserProvided-schema.dtd">

<userprovided>
```

```
<factory>test.readerwritersafe.userprovided.ReaderWriterSafeUserProvidedFactory</fa
ctory>
   <monitor>
      <method name="acquireRead" id="acquireRead">
         <parameterTypes>
         </parameterTypes>
         <wait id="acquireRead_wait">
            <variable>readers</variable>
            <variable>writing</variable>
         </wait>
      </method>
      <method name="releaseRead" id="releaseRead">
         <parameterTypes>
         </parameterTypes>
         <notifyAll id="releaseRead_notify">
            <variable>readers</variable>
            <variable>writing</variable>
         </notifyAll>
      </method>
      <method name="acquireWrite" id="acquireWrite">
         <parameterTypes>
         </parameterTypes>
         <wait id="acquireWrite_wait">
            <variable>readers</variable>
            <variable>writing</variable>
         </wait>
      </method>
      <method name="releaseWrite" id="releaseWrite">
         <parameterTypes>
         </parameterTypes>
         <notifyAll id="releaseWrite_notifyAll">
         </notifyAll>
      </method>
   </monitor>
</userprovided>
```

APPENDIX C

FAIRBRIDGE CASE STUDY

Correct FairBridge Source Code

```
import edu.uta.monitortester.hook.SyncHook;

public class FairBridge
{

   private int nred = 0;

   private int nblue = 0;

   private int waitblue = 0;

   private int waitred = 0;

   private boolean blueturn = true;

   public synchronized void redEnter()
      throws java.lang.InterruptedException
   {
      ++waitred;
      while (nblue > 0 || waitblue > 0 && blueturn) {
         SyncHook.waitThread(this, "redEnter_wait_1");
      }
      --waitred;
      ++nred;
   }

   public synchronized void redExit()
   {
      --nred;
      blueturn = true;
      if (nred == 0) {
         SyncHook.notifyAllThreads(this, "redExit_notify_1");
      }
   }

   public synchronized void blueEnter()
      throws java.lang.InterruptedException
   {
      ++waitblue;
      while (nred > 0 || waitred > 0 && !blueturn) {
         SyncHook.waitThread(this, "blueEnter_wait_1");
      }
      --waitblue;
```

```
        ++nblue;
    }

    public synchronized void blueExit()
    {
        --nblue;
        blueturn = false;
        if (nblue == 0) {
            SyncHook.notifyAllThreads(this, "blueExit_notify_1");
        }
    }

}
```

FairBridge XML Configuration File

```
<?xml version="1.0"?>
<!DOCTYPE userprovided SYSTEM "UserProvided-schema.dtd">

<userprovided>
    <factory>test.fairbridge.userprovided.FairBridgeUserProvidedFactory</factory>
    <monitor>
        <method name="redEnter" id="redEnter">
            <parameterTypes>
            </parameterTypes>
            <wait id="redEnter_wait_1">
                <variable>nblue</variable>
                <variable>waitblue</variable>
                <variable>blueturn</variable>
            </wait>
        </method>
        <method name="redExit" id="redExit">
            <parameterTypes>
            </parameterTypes>
            <notifyAll id="redExit_notify_1">
                <variable>nred</variable>
            </notifyAll>
        </method>
        <method name="blueEnter" id="blueEnter">
            <parameterTypes>
            </parameterTypes>
            <wait id="blueEnter_wait_1">
                <variable>nred</variable>
            </wait>
        </method>
```

```
      <method name="blueExit" id="blueExit">
        <parameterTypes>
        </parameterTypes>
        <notifyAll id="blueExit_notify_1">
          <variable>nblue</variable>
          <variable>waitred</variable>
          <variable>blueturn</variable>
        </notifyAll>
      </method>
    </monitor>
  </userprovided>
```

APPENDIX D

FAIRALLOCATOR CASE STUDY

Correct FairAllocator Source Code

```
import edu.uta.monitortester.hook.SyncHook;

public class BoundedOvertakingAllocator
{

   private int available;

   private int turn = 0;

   private int next = 0;

   private int bound;

   private int overtaken = 0;

   public BoundedOvertakingAllocator( int n, int b )
   {
      available = n;
      bound = b;
   }

   public synchronized void get( int n )
      throws java.lang.InterruptedException
   {
      int myturn = turn;
      ++turn;
      boolean overtakenMe = false;
      while (n > available || overtaken > 0 && !overtakenMe) {
         SyncHook.waitThread(this, "get_wait_1");
         if (next >= myturn + bound && !overtakenMe) {
            overtakenMe = true;
            ++overtaken;
         }
      }
      if (overtakenMe) {
         --overtaken;
      }
      ++next;
      available -= n;
      SyncHook.notifyAllThreads(this, "get_notify_1");
   }

   public synchronized void put( int n )
```

```
    {
        available += n;
        SyncHook.notifyAllThreads(this, "put_notify_1");
    }

}
```

FairAllocator XML Configuration File

```
<?xml version="1.0"?>
<!DOCTYPE userprovided SYSTEM "UserProvided-schema.dtd">

<userprovided>
    <factory>test.fairallocator.userprovided.FairAllocatorUserProvidedFactory</factory>
    <monitor>
        <method name="get" id="get">
            <parameterTypes>
                <parameterType>int</parameterType>
            </parameterTypes>
            <wait id="get_wait_1">
                <variable>available</variable>
                <variable>next</variable>
            </wait>
            <notifyAll id="get_notify_1">
            </notifyAll>
        </method>
        <method name="put" id="put">
            <parameterTypes>
                <parameterType>int</parameterType>
            </parameterTypes>
            <notifyAll id="put_notify_1">
            </notifyAll>
        </method>
    </monitor>
</userprovided>
```

APPENDIX E

BOUNDEDOVERTAKINGALLOCATOR CASE STUDY

Correct BoundedOvertakingAllocator Source Code

```
import edu.uta.monitortester.hook.SyncHook;

public class BoundedOvertakingAllocator
{

   private int available;

   private int turn = 0;

   private int next = 0;

   private int bound;

   private int overtaken = 0;

   public BoundedOvertakingAllocator( int n, int b )
   {
      available = n;
      bound = b;
   }

   public synchronized void get( int n )
      throws java.lang.InterruptedException
   {
      int myturn = turn;
      ++turn;
      boolean overtakenMe = false;
      while (n > available || overtaken > 0 && !overtakenMe) {
         SyncHook.waitThread(this, "get_wait_1");
         if (next >= myturn + bound && !overtakenMe) {
            overtakenMe = true;
            ++overtaken;
         }
      }
      if (overtakenMe) {
         --overtaken;
      }
      ++next;
      available -= n;
      SyncHook.notifyAllThreads(this, "get_notify_1");
   }

   public synchronized void put( int n )
```

```
    {
        available += n;
        SyncHook.notifyAllThreads(this, "put_notify_1");
    }

}
```

BoundedOvertakingAllocator XML Configuration File

```
<?xml version="1.0"?>
<!DOCTYPE userprovided SYSTEM "UserProvided-schema.dtd">

<userprovided>

<factory>test.boundedovertakingallocator.userprovided.BoundedOvertakingAllocatorU
serProvidedFactory</factory>
    <monitor>
        <method name="get" id="get">
            <parameterTypes>
                <parameterType>int</parameterType>
            </parameterTypes>
            <wait id="get_wait_1">
                <variable>available</variable>
                <variable>next</variable>
                <variable>overtaken</variable>
            </wait>
            <notifyAll id="get_notify_1">
            </notifyAll>
        </method>
        <method name="put" id="put">
            <parameterTypes>
                <parameterType>int</parameterType>
            </parameterTypes>
            <notifyAll id="put_notify_1">
            </notifyAll>
        </method>
    </monitor>
</userprovided>
```

REFERENCES

[1] Artho, C. & Biere, A. "Applying static analysis to large-scale, multi-threaded Java programs," In Proc. of the 13th Australian Software Engineering Conference, pp.68–75, 2001.

[2] Barrett, C., Dill, D. & Levitt, J. “Validity Checking for Combinations of Theories with Equality,” In Procs. of the 1st International Conference on Formal Methods in Computer-Aided Design, vol. 1166, pp.187-201, 1996.

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

[4] Carver, Richard H. & Tai, Kuo-Chung. Modern Multithreading, John Wiley & Sons, 2005, ISBN: 0-471-72504-8.

[5] Clarke, E.M., Emerson, E.A. & Sistla, A.P. "Automatic verification of finite-state concurrent systems using temporal logic specifications," ACM Transactions on Programming Languages and Systems, 8(2):244–263, 1986.

[6] Clarke, E.M., Grumber, O. & Peled, D.A. “Model Checking,” MIT Press, USA, 1999.

[7] Cok, D.R., & Kiniry, J. "ESC/Java2 : Uniting ESC/Java and JML. Progress and issues in building and using ESC/Java2 and a report on a case study involving the use of ESC/Java2 to verify portions of an internet voting tally system," Lecture Notes in Computer Science, 3362:108-128, 2005.

[8] Corbett, James C., Dwyer, Matthew B., Hatcliff, John, Laubach, Shawn, Pasareanu, Corina S., Robby, & Zheng, Hongjun. “Bandera: Extracting Finite-state Models from Java Source Code,” In Proc. 22nd International Conference on Software Engineering (ICSE), pp.439-448, 2000.

[9] Demartini, C., Iosif, R. & Sisto, R. “Modeling and Validation of Java Multithreaded Applications using SPIN,” In Proc. of the 4th SPIN Workshop, November 1998.

[10] Detlefs, D., Nelson, G. & Saxe, J.B. "A theorem-prover for program checking," Technical Report HPL-2003-148, HP Systems Research Center, 2003.

[11] ESC/Java2 Website, http://secure.ucd.ie/products/opensource/ESCJava2/.

[12] FindBugs Website, http://findbugs.sourceforge.net/.

[13] Godefroid, P. “Model Checking for Programming Languages using VeriSoft,” In Proc. of the 24th ACM Symposium on Principles of Programming Languages, pp. 174-186, 1997.

[14] Gosling, J., Joy, B., Steele, G. & Bracha, G. “The Java Language Specification, Third Edition”, Prentice Hall, 2005.

[15] Guttag, J., Horning, J., Garland, S., Jones, K., Modet, A. & Wing, J. “Larch: Languages and Tools for Formal Specification,” Springer-Verlag, 1993.

[16] Hamlet, R.G. "Testing programs with the aid of a compiler," IEEE Transactions on Software Engineering, 3(4):279-290, 1977.

[17] Hansen, P.B. “Reproducible Testing of Monitors,” Software Practice and Experience, vol. 8, pp.721-729, 1978.

[18] Hansen, P.B. “The programming language Concurrent Pascal,” IEEE Trans. On Software Engineering, 1(2):199-207, 1975.

[19] Harvey, C. & Strooper, P. “Testing Java monitors through deterministic execution,” In Proc. of Australian Software. Engineering Conference, pp.61-67, 2001.

[20] Hatcliff, J. & Dwyer, M. "Using the Bandera tool set to model-check properties of concurrent Java software," In Proc. of the 12th International Conference on Concurrency Theory, pp.39–58, 2001.

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

[22] Holzmann, G.J. “The model checker SPIN,” IEEE Transactions on Software Engineering, 23(5):279–295, 1997.

[23] Holzmann, G.J. “The SPIN Model Checker,” Addison Wesley, 2004.

[24] Hovemeyer, D. & Pugh, W. “Finding bugs is easy,” http://findbugs.sourceforge.net/docs/findbugsPaper.pdf.

[25] Java PathFinder Website, http://javapathfinder.sourceforge.net/.

[26] Jlint Website, http://artho.com/jlint/.

[27] JML Website, http://www.cs.iastate.edu/~leavens/JML/.

[28] Kim, S. K., Wildman L. & Duke, R. “A UML Approach to the Generation of Test Sequences for Java-based Concurrent Systems,” In Proc. Of the Software Engineering Conference, pp.100-109, 2005.

[29] Lei, Y., Carver, R., Kung D., Hernandez, M. & Gupta, V.. “A State Exploration-Based Approach to Testing Java Monitors,” 17th International Symposium on Software Reliability Engineering, pp 256-265, 2006.

[30] Lichtenstein, O. & Pnueli, A. “Checking that finite state concurrent programs satisfy their linear specifications,” In Proc. of the 12th ACM Symposium on Principles of Programming Langues, pp.97-107, 1985.

[31] Lipton, R.J., DeMillo, R.A. & Sayward, F.G. "Hints on test data selection: Help for the practicing programmer," IEEE Computer, 11(4):34-41, 1978.

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

[33] Magee, J. & Kramer, J. “Concurrency: State Models & Java Programs”, John Wiley & Sons, 1999.

[34] Ma, Y., Offutt, J. & Kwon, Y. “MuJava: An Automated Class Mutation System,” Journal of Software Testing, Verification and Reliability, 15(2):97-133, 2005.

[35] Owre, S., Rushby, J., Shankar, N. & von Henke., F. "Formal verification for fault-tolerant architectures: Prolegomena to the design of PVS," IEEE Transactions on Software Engineering, 21:107–125, 1995.

[36] Paulson., L.C. "Isabelle: the next 700 theorem provers," Logic and Computer Science, pp. 361–386, Academic Press, 1990.

[37] Quielle, J.P. & Sifakis, J. “Specification and verification of concurrent systems in CESAR,” In Proc. of the 5th International Symposium on Programming, vol. 137, pp.337-351, 1981.

[38] Savage, S., Burrows, M., Nelson, G. & Sabalvarro, P. “Eraser: A Dynamic Data Race Detector for Multithreaded Programs,” ACM Trans. On Computer Systems, 15(4):391-411, 1997.

[39] The Java Virtual Machine Specification, Second Edition, http://java.sun.com/docs/books/jvms/second_edition/html/VMSpecTOC.doc.html.

[40] Vardi, M.Y. & Wolper, P. “An automata-theoretic approach to automatic program verification,” In Proc. of the 1st Symposium on Logic in Computer Science, pp.322-331, 1986.

[41] Visser, W., Havelund, K., Brat, G. & Park, S. "Model checking programs," In Proc. of the 15th International Conference on Automated Software Engineering, pp.3-12, 2000.

[42] Wildman, L., Long, B. & Strooper, P. "Dealing with Non-Determinism in Testing Concurrent Java Components," 12th Asia-Pacific Software Engineering Conference (APSEC), pp.393-400, 2005.

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

BIOGRAPHICAL INFORMATION

Andres Yanes received his M.S. in Computer Science and Engineering from the University of Texas at Arlington in May 2007 and his B.S. in Biomedical Engineering from Boston University in May 1997. His research interests include software testing and software engineering.