TESTING JAVA MONITORS BY STATE SPACE EXPLORATION

by

MONICA HERNANDEZ

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 2006

ACKNOWLEDGEMENTS

I would like to thank my husband for all his support and understanding. Alex, you have been my major source of encouragement and motivation and you never complained about all the time this period of time took away from us. You were also always trying to push me to try harder and get to the final goal without hesitation. My mom, dad and sister were also key to my success, with their love and encouragement that always helped me get through the hard moments.

I would also like to thank my supervisor, Jeff Lei, for the direction he provided and for being so willing to help all the time, especially with the time constraint I had. I really enjoyed our interesting discussions with you and my partner Vidur, about the approach and implementation challenges.

Working full time and studying at the same time was challenging, and this would have not been possible without the support from my company American Leather, who not only supported me economically but also made it easier to accommodate to the classes schedule. Finally, thanks to my friends, and anyone else that contributed to this great achievement.

November 21, 2005

ABSTRACT

TESTING JAVA MONITORS BY STATE SPACE EXPLORATION

Publication No. _____

Monica M Hernandez, M.S.

The University of Texas at Arlington, 2006

Supervising Professor: Dr. Jeff Lei

Java monitors are classes that are intended to be accessed by multiple threads at the same time. Detecting synchronization faults in Java Monitors is considerably more challenging than testing regular classes, due to the inherent non-determinism of concurrent programs. This thesis proposes a state based exploration approach to testing Java monitors. This approach consists of exploring the state space of a Java monitor in a depth-first manner, dynamically building test sequences, which are comprised by the states explored along each path. Moreover, threads are introduced on the fly during the exploration of each path, based on several rules for simulating race conditions that may occur when more than one thread is trying to access the monitor at the same time. A

prototype tool called *MonitorExplorer* was developed, and case studies were reported in which the tool was applied to several Java monitors as well as their mutants. The experimental results indicate that the approach is effective in detecting synchronization faults due to the existence of race conditions.

TABLE OF CONTENTS

LIST OF ILLUSTRATIONS

Figure Page

LIST OF TABLES

Table Page

CHAPTER 1

INTRODUCTION

Multithreaded programming has become a key to modern software development, because it offers greater computational efficiency allowing some threads to execute certain tasks while others are waiting for some resource. Even more, there are problem domains that are inherently concurrent, and therefore can be solved more effectively using multiple threads. Web applications, for example, demand multithreaded environments that can serve multiple client requests. However, concurrent programming presents some challenges that sequential programming doesn't have, such as non-determinism. The results of a sequential program will remain constant given a fixed set of input and operational parameters. Parallelism, however, often leads to non-determinism, so results depend on the order of execution. This is due to synchronization and communication among the threads, and race conditions. Race conditions are "a situation in which the final result of operations being executed by two or more units of execution depends on the order in which those units of execution execute. For example, if two units of execution A and B are to write different values VA and VB to the same variable, then the final value of the variable is determined by the order in which A and B execute" [8]. The main focus of this thesis is on detecting problems caused by race conditions, because we believe that a lot of synchronization

issues are due to problems when more than one thread are trying to access the monitor at the same time.

Monitors are a synchronization mechanism that encapsulates the representation of a shared resource and provides operations that are the only way of manipulating it. Java is a very popular language that provides a Monitor implementation for thread synchronization, referred to in this thesis as Java Monitors, which are a particular case of concurrent programs.

1.1 Java Monitors

A Java monitor is a class that defines one or more synchronized methods, i.e., methods whose signatures contain the keyword synchronized. The Java runtime automatically enforces mutual exclusion on the synchronized methods in a Java monitor. In general, a monitor ensures that at most one thread can be active within the monitor, so the main reason for having monitors is to maintain data integrity on shared data. Figure 1 shows a graphical view of a monitor for the Bounded Buffer problem. This monitor has two synchronized methods, *deposit()* and *withdraw()*. A thread that executes the method deposit is called a producer, and a method executing withdraw is called a consumer. The requirement for this problem is that when the buffer is full or empty, the producer or consumer must be blocked, respectively.

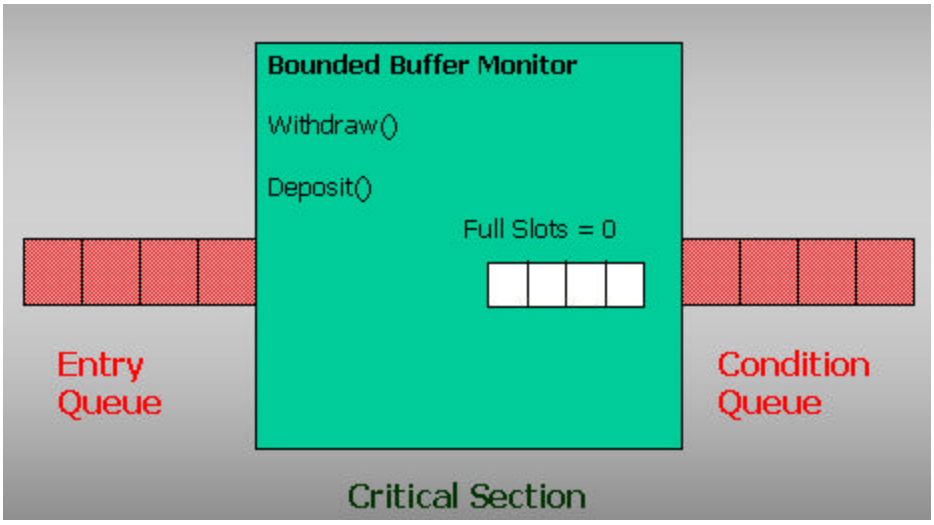

Figure 1.1 Java Monitor Example – Bounded Buffer

For each Java monitor, there are three main components:

- Entry Queue (EQ): This queue controls threads trying to access the shared resource. If a thread calls a synchronized method while another thread is executing inside the monitor, the calling thread must wait on the entry queue of the monitor until it gains the lock.
- Critical Section (CS): A thread must be inside the critical section in order to execute a synchronized method. Java runtime automatically enforces mutual exclusion, so only one thread can be inside the critical section.
- Condition Queue (CQ): Only a thread that is inside the critical section can go to the condition queue, by executing the operation *wait()*. This operation is used when the thread needs to block itself until another thread signals it with the operation *notify()/notifyAll()*. When a thread executes wait(), it leaves the critical section and goes to the condition queue, which allows other threads in

the entry queue to enter the CS. When a thread executes notify() (or notifyAll()), it awakens one (or all) of the threads blocked in the condition queue, if the queue is not empty, and then continues to execute inside the CS. An awakened thread does not immediately re-enter the CS. Instead, it joins the entry queue and thus competes with other threads trying to enter/re-enter the CS. Note that according to the Java specification, notify() does not necessarily preserve First-Come-First-Serve semantics, i.e., it may not awaken the longest waiting thread.

Fig. 1.2 shows the code for a Java monitor that solves the bounded buffer problem.

Many approaches have been developed to test regular classes, in which a test is a sequence of method calls that are issued by a single test driver thread. These approaches, however, cannot be directly applied to a Java monitor, which is intended to be accessed by multiple threads simultaneously, instead of just one thread like it is intended for a regular object. Therefore, in order to replicate possible scenarios in which a Java monitor may be used, it is necessary to create more than one thread in a test. Further more, if a single thread is used to test a Java Monitor, the whole program would be blocked when a thread executes a synchronization operation like wait(). Consequently, Java Monitors testing requires more than one thread, which raises new issues:

```
public class BoundedBufferExample{
    public int fullslots=0;
    private int capacity = 0;
    private Integer[] buffer = null;
    private int in = 0, out = 0;

    public synchronized void deposit(Integer value) {
        while(fullslots == capacity) {
            try{
                wait();
            }catch(Exception e){};
        }
        buffer[in]=value;
        in=(in + 1)%capacity;
        if(this.fullslots++ == 0) {
            notifyAll();
        }
    }
    public synchronized Integer withdraw() {
        Integer value=new Integer(0);
        if (fullslots == 0){
            try{
                wait();
            }catch(Exception e){};
        }
        value = (Integer)buffer[out];
        out = (out + 1) % capacity;
        if (fullslots-- == capacity){
            notifyAll();
        }
        return value;
    }
}
```

Figure 1.2 Java Monitor that Solves the Bounded Buffer Problem

- Number of threads: As mentioned above, one thread is not enough to test java monitors. Even more, many synchronization faults can only be detected when a certain minimum number of threads interact, but usually the number of threads necessary is not known *a priori*. This raises the issue of how many threads are

needed in order to find problems with the monitor, and how and when those threads are introduced.

- Non-deterministic behavior: Since a Java monitor by nature exhibits non-determinism, the behavior of each test executed can be non-deterministic as well. Therefore, how can the behavior of each test be specified, and how can the execution be controlled so that the desired behavior is exercised for each test?
- Thread manipulation: In order to control the execution so that the desired behavior is exercised, threads must be manipulated in a way that the *Java's standard virtual machine* does not handle. As mentioned above, a notify operation in a Java Monitor does not necessarily awakens the thread that has been waiting the longest (It is not FIFO). The execution of each test sequence will require that specific threads are signaled in order to create the different scenarios, so a method to manipulate the notify operation and yet simulate the Java Monitor must be developed.

This thesis presents an approach to solve the above issues, which mainly consists of systematically exploring the state space of a Java monitor. Each path explored is a dynamically built test sequence, because the paths and threads to be introduced along the path are not known a priori. Each path is comprised of the states from the initial state to an end state, which is the one where the exploration backtracks. Each state is a decision making point, where new threads may be introduced on the fly or other operations executed, following rules to try to simulate race conditions and therefore

detect synchronization problems that may occur when multiple threads try to access the same monitor at the same time. Because the paths are executed at the same time they are being generated, the processes of test generation and test execution are interleaved, which differs from other approaches developed over the years. In order to characterize and control the execution, we use transitions that indicate what is next in the exploration process, i.e. whether new threads are introduced and which methods to execute. Every time a transition is executed, an Abstract States is created that encapsulates the information about the monitor that may affect its behavior. A state is also generated every time a thread executes a synchronization operation which is intercepted by a wrapper component that stops the execution of that thread until the state is evaluated. The exploration of a path comes to an end when a duplicate state is found, at which point the exploration backtracks to follow other paths until all paths are explored. Each state is evaluated during the exploration process, in order to check the requirements which are validated by evaluation conditions provided by the user. We implemented a prototype tool that is described further in the following chapters. The results obtained from our experiments with five different monitors show that our tool can effectively detect common synchronization problems.

The rest of this document is organized as follows: Chapter II briefly surveys related work. Chapter III presents our state space exploration based approach and the implementation of the prototype tool. Chapter V reports the results of our experiments with three classic monitors: Bounded Buffer, Safe Bridge and Readers and Writers. Finally, Chapter VI provides conclusions and future enhancements.

CHAPTER 2

RELATED WORK

2.1 Testing Concurrent Pascal Monitors

Brinch Hansen [5] proposed a method where the user specifies a set of preconditions and then builds a sequence of monitor calls to exercise the preconditions. The methodology has the following steps:

- Step 1: The tester identifies a set of preconditions that will cause each branch of the operation to be executed at least once
- Step 2: The tester constructs a sequence of monitor calls that will exercise each operation under each of its preconditions.
- Step 3: The tester constructs a set of test processes that will interact exactly as defined above.
- Step 4: These processes are scheduled by means of a clock. The test program is executed and its output is compared with the predicted output.

This work only works for Pascal Monitors, so it was later extended by Craig Harvey and Paul Strooper [1] so it could be applied to Java Monitors. This approach requires a lot of manual intervention by the user in order to create the preconditions and test sequences. A clock is used to synchronize the methods, but there is no way to detect when threads are done with a call so it may not find all the errors in the programs. Our approach uses state space exploration to dynamically build the test sequences and it

executes them at the same time, so the type of errors that can be found could be different. Furthermore, our tool has a higher degree of automation.

2.2 Constraint Based Approach

Carver and Tai [7] generalized Brinch Hansen's technique for synchronizing threads during testing and showed how to apply their technique to monitors.

This methodology proposes these steps:

Step 1: Derive a set of validity constraints from a specification of the program

Step 2: Performing non-deterministic testing, collecting the results to determine coverage and validity

Step 3: Generate additional test sequences for paths that were not covered, and performing deterministic testing for those test sequences.

This method requires a specification and it does not have tool support so it is hard to apply in practice. The main contribution of this approach is the definition of constraints that can reduce the state exploration based on observable events only and thus avoid state explosion.

2.3 Model Checking

A model is a simplified representation of the real world, which includes only those aspects relevant to the problem being resolved. Model checking has been used to automatically test interactive programs written in a constraint based language [4][6]. The method uses an algorithm to systematically generate all possible behaviors of such a program, and these behaviors are then monitored and checked against user-specified safety properties. This approach is based on state space exploration techniques, like

ours is. However, they either directly explore the state space of a concurrent program or extract an abstract model from the program and then explore the abstract model using a formal methods tool. The issue that remains unresolved in this approach is the state explosion that could happen during exploration. On the other hand, our approach introduces threads on-the-fly, as needed, during state space exploration, whereas model checking approaches assume programs are closed so number of threads have to be known a priori.

2.4 Java Monitors Testing

As mentioned above in Section 2.1, the work by Brinch Hansen [5] was extended in 2001 [1][2] to apply method to Java Monitors. Java Monitors are different than Pascal Monitors in that they do not have condition variables so all the threads waiting on different conditions wait in the same condition queue. Therefore, branch coverage is not enough because a while loop has to be used for all threads to check the condition before re-entering the critical section. Thus, the first step requires that the identified preconditions not only cause every branch to be executed but also cause every loop to be executed zero times, one time and more than one time. The other characteristic of Pascal Monitors that Java does not provide is the "immediate resumption requirement", which guarantees that the thread waiting the longest in the condition queue is the one notified.

Long, D. Hoffman, and P. Strooper [1] introduce tool support for unit testing concurrent Java components in 2003 . Their tool ConAn (Concurrency Analyser), automates the third step in Brinch Hansen's method [5]. In addition, they reduced the

original method to three steps. With ConAn, the tester specifies the sequence of calls and the threads that will be used to make those calls. It then generates a test driver that controls the synchronization of the threads through a clock and that compares the outputs against the expected outputs specified in the test sequence. The tables 2.1 and 2.2 show an example of some of the conditions and a test sequence that would be defined for the Bounded Buffer using this approach.

Table 2.1 Example of Test Conditions for the Bounded Buffer

Method	Condition	Condition description
withdraw()	C_1	0 iterations of the loop
withdraw ()	C_2	1 iteration of the loop
withdraw ()	C_3	Multiple iterations of the loop
deposit()	C_4	0 iterations of the loop
deposit ()	C_5	1 iteration of the loop
deposit ()	C_6	Multiple iterations of the loop

Table 2.2 Example of Test Sequence for the Bounded Buffer

Time	Thread	Call	Output	Conditions	Call Completion
1	T_1	deposit("a")	-	C_5,C_8	[1,2)
2	T_2	deposit("b")	-	C9,C_6	[3,4)
3	T_3	withdraw()	'a'	-	[3,4)
4	T_4	withdraw()	'b'		[4,5)

Even though this approach automates the generation of the test driver, the user still has to create the conditions and the test sequences, which could be difficult and error prone. On the other hand, our approach can be automated to generate and execute the test sequences. In addition, this approach does not entirely address race conditions, which is the main focus of this thesis.

CHAPTER 3

STATE SPACE BASED EXPLORATION APPROACH

The algorithm takes an initial state of the monitor to be tested specified by the user. Different initial states can be used to execute the tool, usually ones that use the lower and upper bounds of the monitor. The algorithm generates an abstract state using the method getAbstractState() which collects relevant information about the monitor useful to generate the transitions and to evaluate the monitor. Then it uses the method getEnabledTransitions() which finds all the possible transitions for that abstract state. A transition represents an action that indicates the operation to be performed, such as introduce a thread or execute a thread of a certain type. A thread type is the method that the thread executes. For example, one transition could be introduce a thread of type consumer, or execute a thread of type producer which is in the head of the entry queue. Each transition is then executed by a driver call executeMethod() or introduceThread() depending on the transition, which will get a thread from a thread pool and execute a method if necessary. Each state is evaluated against the conditions defined by the user and for uniqueness so no duplicate states are explored. If the state is not valid, the path is cancelled and the algorithm backtracks to the prior branching point. The algorithm uses two main structures to keep the stack of transitions that have been executed and the visited states to be able to check for duplicates. Each path explored (from an initial state to an end state which is one where the exploration backtracks) can be considered a

dynamically built test sequence, since threads are introduced on the fly as needed by the exploration process. The threads are introduced in a way that race conditions would be created in order to detect synchronization problems. An important aspect of our approach is that the test sequences are generated at the same time they are executed, based on a depth-first exploration of the states. The rest of this chapter describes how this approach uses the Abstract State and the transitions to control the behavior of the execution and evaluate the Java Monitor.

3.1 Abstract State

Appropriate state abstractions are necessary to ensure that the exploration of the state space of a Java monitor terminates. An abstract state represents the relevant information about the monitor state, needed for the test process. The abstract state values will be specific to each monitor being tested, but the structure has to be flexible and generic to accommodate any monitor state with any number of threads introduced. The abstract state is used to:

- Determine enabled transitions
- Check for duplicate states to ensure exploration terminates
- Check for invalid states

The attributes of the abstract state are:

- Entry queue: The ID and type of the thread in the head of the entry queue. The approach will always try to create race conditions in order to identify problems, so what matters is whether there is a thread in the entry queue and if so, what type (i.e. consumer or producer)

- Critical Section: The ID and type of the thread currently executing in the critical section.
- Condition queue: A lot of different threads could be in the condition queue, but for evaluation purposes all we need to know is the types of the threads in this queue.
- Data members/attributes values: The values of the attributes in the monitor. This is a string representing the current state in terms of thread types being in the monitor. It is also an abstraction of the current value of the data members with respect to a lower or upper bound limit.

The first 3 values of the abstract state can be generated automatically by the tool and will always be calculated the same way for any Java Monitor. However, the last one, data members, depends on the specific monitor being tested so it should be implemented by the user in a User Implementation class called UserAbstractState which extends the AbstractState class. Table 3 shows an example of the possible values of the Abstract State for the Bounded Buffer.

Table 3.1 Bounded Buffer Abstract State for Data Members

Data Members	FullSlots	Valid
0	FullSlots=0	Y
0-N	0<FullSlots<N	Y
N	FullSlots=N	Y
N++	FullSlots>N	N
0--	FullSlots<0	N

Since the user has to define the Abstract State for the Monitor being tested, we describe some guidelines to make sure the testing is successful. The data members abstraction should be independent of the number of threads, because this number is not known a-priori and it won't affect the behavior of the monitor. The user can also use thread types based on the method being executed, for example consumer and producer in the Bounded Buffer example. We only consider data members that may affect the synchronization behavior of a monitor. A key observation is that a data member affects the synchronization behavior of a monitor if it is referenced in a branching statement which leads to paths that may display different synchronization behavior. Therefore, the abstract values of a data member can be identified by partitioning the domain of the data member into intervals that lead to those different paths. In the example in Table 3 the value 0 indicates fullSlots = 0, 0 – N indicates the value falls between 0 and N (N is

the buffer size) and N represents the fullSlots = N. In the Bounded Buffer example, the critical section abstraction could be "", deposit or withdraw. The first case will indicate that the critical section is empty, and the other two indicate which type of thread is inside the monitor. For the condition queue the thread types (deposit, withdraw) can be used as well to specify the state of the monitor in that queue. The number of threads in the condition queue of each type is not relevant, but for certain synchronization faults it is important to have more than one thread in the condition queue, so a plus sign can be used to specify that more than one thread of that type is waiting. For example, the state *deposit+* in the condition queue specifies that there is more than one deposit thread in the condition queue, and *deposit* indicates that there is only 1. If both producers and consumers are in the condition queue, the state would be, for example, "*deposit, withdraw*". See the Appendix section for examples of Abstract States for each of the monitors used in the experiments.

3.2 Transitions

The controller characterizes the behavior to be executed by the driver using transitions. A transition represents an action that will change the state of the monitor to the next state in the path. A transition could be to execute a specific method with a thread already in the entry queue, or to introduce a thread either to progress the execution (when no threads are in the entry queue or critical section) or to create a race condition in the middle of a notify/notifyAll operation. A new thread needs to be introduced in the following two cases:

- If the entry queue and the CS are both empty at the current state, then for each monitor method, we will introduce a new thread to execute the method. The motivation for this rule is that otherwise we cannot proceed any further with the exploration as all the existing threads are blocked. Since the introduced threads compete to enter the CS, state exploration will explore the possibility that for each synchronized method, a new thread that executes the method wins the competition. Note that this rule can always be applied to an initial state, where no thread have been introduced yet. Also note that the CS becomes empty when the thread inside the CS exits, which can be due to the execution of wait() or due to reaching the end of a synchronized method.
- If the entry queue at the current state is empty, and the next operation to be executed by the thread inside the CS is a notify operation, then for each synchronized method of the monitor, we will introduce a new thread to execute the method before the notify operation is executed. The motivation for this rule is as follows. Recall that when a notify operation is executed, a thread T in the condition queue will be moved to the entry queue and will compete with other threads to reenter the monitor. The introduction of a new thread T' for each synchronized method places T' in the front of the entry queue. Thus, state exploration will explore the scenario that T loses the competition to T', as T will enter the entry queue after T' when we execute the notify operation.

3.3 Implementation

The main modules of the application are the controller module (executes algorithm), the execution module (driver) which is the one that creates the threads and executes the methods in the threads, and the evaluation which is what tests conditions that should be met by the application. Since we are testing concurrent programs multiple threads have to be created, so the main thread executes the monitor explorer and it introduces new threads as needed (Monitor Threads). Every time a thread executes a synchronization operation (i.e. wait, notify) the Monitor Wrapper intercepts that operation to update the state and notify the main thread via the Communication Monitor. The user of this tool needs to implement certain functions to evaluate the specific java monitor, such as the abstract state (data members abstraction) and the evaluation function that tests the different conditions. Each module will be described further below. Figure 3.1 illustrates the Packages structure used in the implementation.

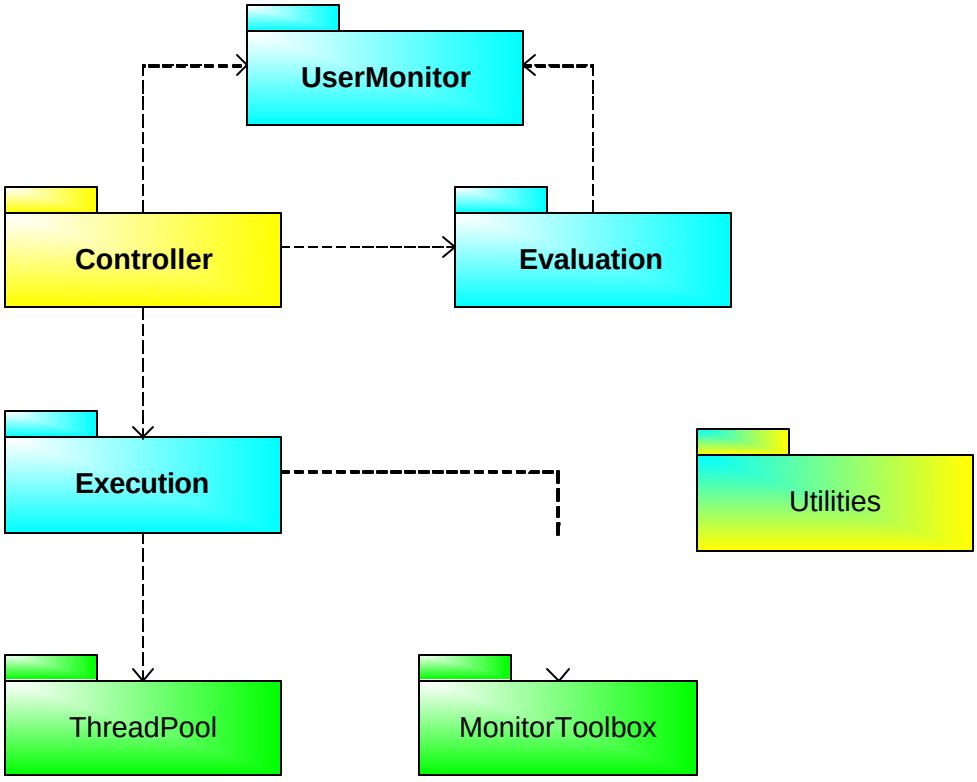

Figure 3.1 Implementation - Packages Structure

3.3.1 Sequence Diagram

The diagram below (Figure 3.2) describes how the different components interact with each other. Each component is described in the subsequent sections.

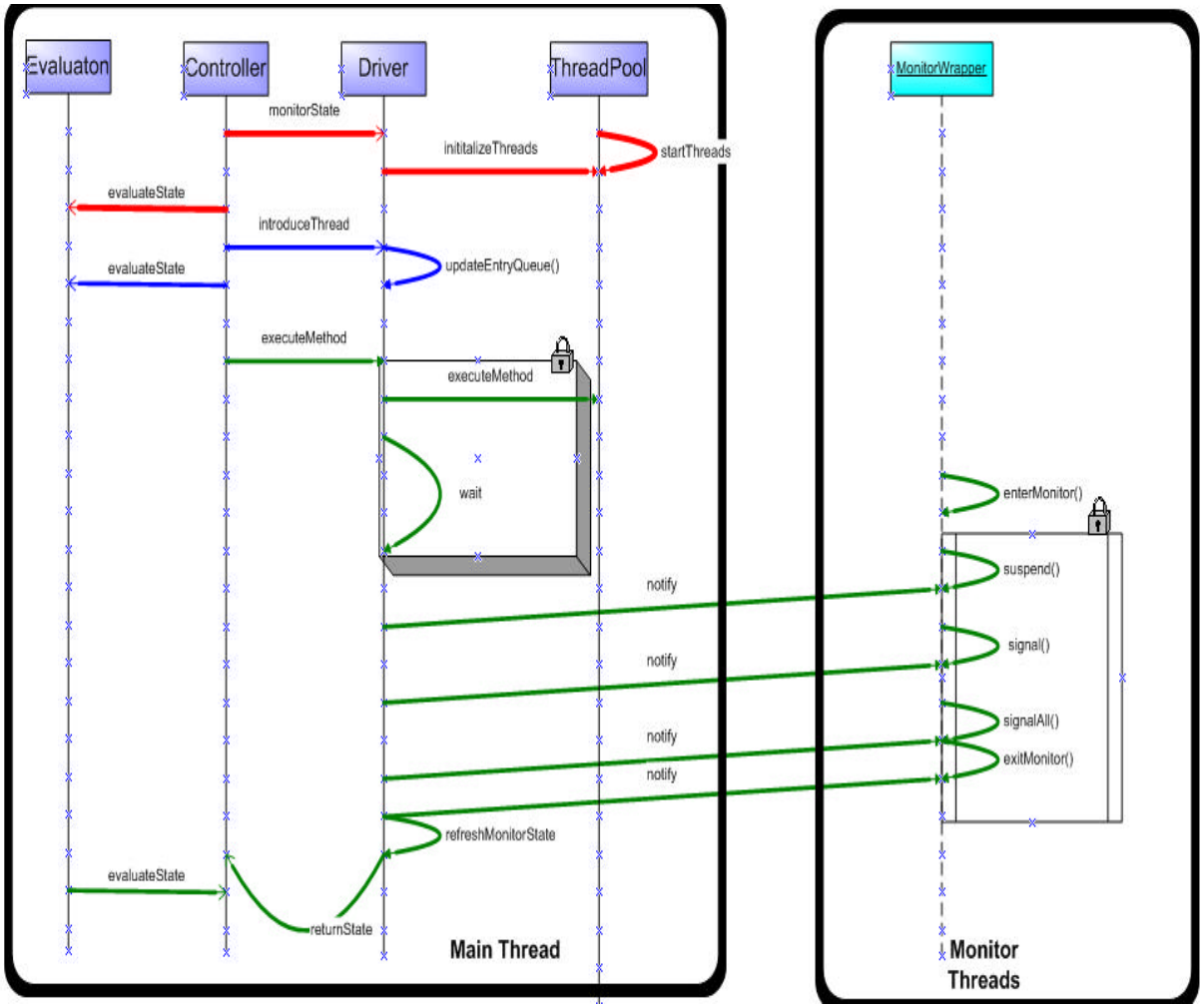

Figure 3.2 Implementation - Sequence Diagram

Sequence Example for Bounded Buffer:

- Initial state is created (no threads have been introduced so entry queue and critical section are empty) EQ="", CQ="", CS="", Data members="0"
- Controller generates transition "introduce withdraw"
- Driver creates a thread and places it in the monitor's entry queue (book-keeping) EQ: "" => withdraw (at this point a method has not been executed with that thread)
- Driver returns control to Controller and returns new Monitor State
- Controller creates new AbstractState based on state returned by Driver, checks for duplicates, evaluates state (using user's evaluation function) and generates new enabled Transitions (based on new state)
- Controller generates transition "execute withdraw"
- Driver:
    - Move withdraw from Entry Queue (withdraw => "") to Critical Section ("" => withdraw)
- Executes the method withdraw with introduced thread (Reflection API)
- Waits for first synchronization operation to be intercepted by wrapper => wait
- Wrapper updates state:
    - Moves from Critical Section (withdraw => "") to Condition Queue ("" => withdraw)
    - Stops execution and returns control to Controller
    - Controller makes decision and so on…

3.3.2 Controller

This modules implements the algorithm to explore the states of the monitor, and it controls the transitions to be executed and threads to be introduced/notified by using the Execution module. This module receives an input file from the user specifying the monitor to be tested and information about that monitor (number of methods, data members/attributes, etc.).

3.3.3 Execution/Driver

The driver creates and manages the execution environment according to the requests from the controller. It interacts with the thread pool to introduce threads and execute methods with the threads. It also uses the Reflection API to be able to execute objects and methods that are not known *a priori*, and to get the actual values of the data members of the monitor at run time.

When the driver executes a method, it waits until the thread reaches a synchronization method at which point it updates the monitor state and notifies the controller that the monitor state has changed before it continues with the execution. The main challenge for this module was to find a way to control the execution of the threads to be able to fulfill the controller requests. In order to create the test conditions necessary to detect problems such as race conditions, some actions need to be performed by the controller depending on the state before the thread can even continue the execution. Therefore, it is the execution module's job to ensure that the execution is "frozen" until a decision and any proper actions are taken by the controller with other threads. This is mainly achieved using two components: the MonitorWrapper class,

which simulates the behavior of a monitor in order to update the monitor state, and to give feedback to the controller based on what the monitor is theoretically going to do when the actual synchronization operation is performed. The second component is the Communication Monitor which allows the monitor threads and the main thread to communicate to ensure synchronization between the application threads. The Communication Monitor is used to block the Main Thread (i.e. Controller/Driver) until the Java Monitors reaches a synchronization operation such as wait() or notify(), and to block the monitor threads to prevent them from continuing the execution until the Controller makes a decision on what the next transition is.

3.3.4 Monitor Wrapper & Monitor Toolbox

Since we need to control the execution of the monitor in order to execute certain test sequences we had to use a monitor implementation called Monitor Toolbox which simulates a java monitor but also allow us to control the way the monitor behaves, such as notifying a specific thread instead of a random thread like the Java Monitor does. We implemented a Wrapper class that uses the toolbox called MonitorToolboxWrapper which takes care of all the book-keeping. This is all the logging of the monitor state before the actual synchronization operation happens. The wrapper is the main mechanism through which we can know what is going on in the execution environment in order to control it. The operations implemented in the wrapper perform. book-keeping and calls back to the controller/driver module before the actual synchronization operation is executed. The synchronization operations in the Java Monitor being tested are replaced by the ones implemented in the wrapper so that the execution of those calls

can be monitored and controlled. Every time an operation in the wrapper is executed by a thread the Monitor State is updated so that the driver can return that information to the controller. The controller uses that state information to create the Abstract State and generate the enabled transitions based on the current state of the monitor.

These are the methods implemented by the wrapper, that allows the tool to intercept synchronization points for the threads and therefore control the execution:

- enterMonitor: Operation that allows a thread to enter the critical section
- suspend: This operation wraps the wait() call.
- Signal/signallAll: This operation wraps the notify()/notifyAll() call. Threads may be introduced dynamically in order to create a race condition between new threads and the thread that is being notified. When the execution is being re-played during the exploration process, the wrapper may have to signal a specific thread in the condition queue instead of a random one like the standard implementation of the Java Monitor does.
- ExitMonitor: Releases the critical section

CHAPTER 4

EVALUATION MODULE

This module is responsible for testing the conditions specified by the user in order to detect problems. This module uses all the information provided by the controller and driver modules in order to test if the monitor requirements are being met. Since the requirements for each java monitor can be different, the user needs to implement certain functions that use pre-defined classes in the Evaluation module. The structure of the classes in this module is as follows:

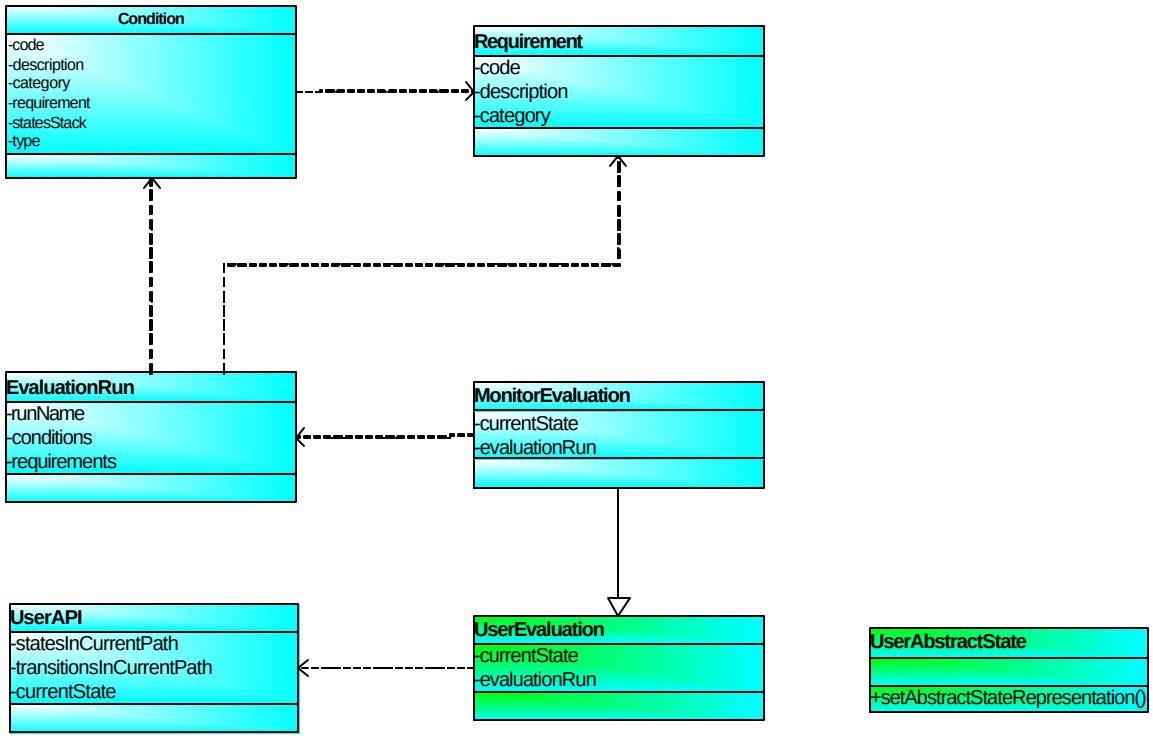

Figure 4.1 Evaluation Module Class Diagram

In order to follow Requirements Engineering practice, the evaluation module was designed in a way that the user starts by specifying what the requirements of the monitor are, so he/she can then create one or more test conditions that will check whether those requirements is met or not. It is very important to note that the conditions defined by the user should test the requirements of the monitor, not a specific implementation of the monitor.

4.1 Evaluation Classes

Requirements: Each requirement has an identifier and a description.

Example:

–Code: 01

–Description: If buffer is full Producer cannot produce

Conditions: The conditions can be specified as being ERROR or WARNING, depending on the developer's criteria. That way all kinds of conditions can be checked instead of only those that are considered errors. The main difference between an error and warning condition is that a warning condition does not stop the test sequence being executed (current path) whereas an error condition does. Each condition can be associated to a requirement so when the condition is found during execution the requirement that is being violated is displayed. With this approach, several conditions can be created that are associated to the same requirement, since there are usually multiple ways to test the same requirement.

Example:

–Code: 01

–Type: Error

–Description: “Data value exceeds upper bound value”

This example shows a condition of the number of buffer full slots being greater than the size of the buffer, which could mean that the producer was allowed to produce when the buffer was full. This condition violates the requirement example shown above “If buffer is full Producer cannot produce”.

EvaluationRun: An evaluation run has a name that identifies it, and a set of conditions that are found during execution. When an error condition is found the current path is interrupted but the other paths or test sequences are still executed, so more conditions could be found. This object can provide the user with a list of all conditions found.

UserAPI: This class provides an interface for the user to access information about the monitor (states, transitions, etc.) in order to test the conditions that will determine whether the monitor is in an invalid state. The main objective of this API is to hide any implementation details of this tool and let the user use generic methods that make more sense to him/her instead of having to understand how this tool was built to access the objects and methods.

MonitorEvaluation: This is an abstract class that contains two abstract methods which the user has to implement in the UserEvaluation class:

- setRequiremetns(): The user uses this method to create the requirement objects to be associated to conditions.
- evaluateMonitor(): This method will test for all the conditions defined by the user based on the pre-defined requirements. It receives a input parameters the

stack of transactions and the stack of all visited states. The user can use these two parameters to initialize the userAPI provided by this module, so he/she can then use the API methods to get information about the monitor in order to check the conditions. This is the most important method in the Evaluation module, since it is the one that uses all the other objects/ methods to evaluate the monitor based on the current state, transitions executed, states visited, etc.

UserEvaluation: Class where the user implements the methods described above.

When a condition is found the stack is printed to a file as a counter scenario for the user to know the sequence of events that it took to get to that condition. Figure 6 shows an example of code that uses the userAPI to get the current state of the monitor (the method evaluateMonitor is called for each state explored) and then uses that state to see if the data member abstraction (in this case fullSlots) indicates that the number of full slots is greater than the size of the buffer (N++), which represents an error condition.

```
currentState = userAPI.getCurrentState();
String abstractStateString = userAPI.getAbstractStateRepresentation();

if (abstractStateString.equals("N++")){
    Condition condition = new Condition("upper_bound", "01", EvaluationRun.TYPE_ERROR, statesInCurrentPath, "Data v
    condition.setRequirement(evaluation.getRequirement("01")); //associate this condition with a predefined requir
    debug.println("Requirement: " + condition.getRequirement().getDescription());
    evaluation.addCondition(condition);
    errorCodes.set(0, Boolean.TRUE);
    validState = false;
```

Figure 4.2 Example of UserEvaluation Class that Evaluates the Bounded Buffer

CHAPTER 5

EXPERIMENTS

5.1 Steps to Test a Java Monitor with our Tool

The objective of our experiments are to confirm that our approach is effective at detecting synchronization faults in Java Monitors and measure the performance of our tool to make sure that the exploration process is able to finish in a reasonable amount of time. The steps that have to be taken to test a Java Monitor with our tool are:

- Change monitor class to use wrapper methods: The user must change the Java Monitor to replace the calls to synchronization operations such as wait or notify/notifyAll to suspend and signal/signalAll, respectively. The methods enterMonitor() and exitMonitor() at the beginning and the end of each method replaces the *synchronized* keyword.
- Provide initializeMonitor function: The user may want t test the monitor with different values. For example, the Bounded Buffer could be initialized with a full buffer (i.e .buffer size = 10 and fullSlots =10) to test producer threads or with an empty buffer (i.e. buffer size = 10 and fullSlots = 0) to test consumer threads. This function needs to be incorporated to the Controller module in order to be used by the tool.

- Define requirements: The user should define the requirements to be tested for the monitor. This is, how the monitor is expected to behave.
- Define Abstract State and evaluation function using the UserAPI in the evaluation module. The Abstract State defined by the user determines the number of unique states that are explored. The user has to consider that a more detailed abstraction will allow for more detailed evaluation but more states will be explored. The evaluation function is called for each state explored and it will return error if an *ERROR* condition is found.
- Create input file (monitor to be tested, etc.): The user provides a file that contains the information about the monitor to be tested, such as monitor name, methods, etc.

5.2 Mutants

A series of mutants were created in order to evaluate the fault detection effectiveness of our tool, which is measured using the number of mutants killed. Each mutant introduces an error in the code. They were defined based on common programming mistakes using Java Monitors. The mutants were created using the following mutations:

• If a *while* loop contains a *wait* operation, then replace the *while* loop with an *if* statement. This operator simulates the user error that a thread that is awakened due to a *notify* or *notifyAll* operation does not re-check a condition when it is supposed to.

• Replace a *notifyAll* operation with a *notify* operation. This operator simulates the programming error that only one thread is awakened from the condition queue when all the threads are supposed to be awakened.

• Remove a *wait*, *notify*, or *notifyAll* operation.

• Replace a Boolean operator with its negation if the operator appears in a branching statement that contains a *wait*, *notify*, or *notifyAll* operation.

• Replace a relational operator with a different relational operator in a branching statement that contains a *wait*, *notify*, or *notifyAll* operation.

• If a Boolean expression appears in a branching statement that contains a *wait*, *notify*, or *notifyAll* operation, and if the expression only contains a single Boolean variable, then replace the Boolean variable with its negation.

Note that the above mutations represent some commonly found programming errors. To avoid a masking effect, only a single change was made to each mutant.

5.3 Monitors Used for Testing

The experiments were made with three classic monitors, to which some mutants were applied (based on the list of mutants described above) in order to detect problems. This section lists the monitors tested and a table with the results. More details on the experiments results and the code used can be found in the Appendix section.

BoundedBuffer: A solution to the Producer/Consumer problem. When the buffer is full, a producer must wait for a consumer to withdraw an item. When the buffer is empty, a consumer must wait for a producer to deposit an item.

• *SafeBridge*: A solution to prevent collisions on a single-lane bridge. Cars coming from different directions cannot access the bridge at the same time.

• *FairBridge*: A solution to prevent collisions on a single-lane bridge. This solution guarantees no starvation. That is, cars from both directions get a fair chance to access the bridge.

5.3.1 Bounded Buffer Experiment Example

This section describes an example of one of the experiments executed with the Bounded Buffer, where the mutant while => if (*while* statement was replaced with *if*) was applied to the method *withdraw()*. Remember that the problem description for the Bounded Buffer is that when the buffer is full, a producer must wait for a consumer to withdraw an item. When the buffer is empty, a consumer must wait for a producer to deposit an item. The initial state for this experiment was the buffer size equal to 10 and the *fullSlots* data member equal to 0. This is, the buffer is empty. The synchronization fault in this experiment is illustrated by the figure 5.1. This is the case where the developer uses an *if* statement instead of a *while*, which causes a problem when awakened threads go from to the condition queue to the entry queue, but another thread that is competing to get inside the monitor is able to gain access first. Therefore, the winning thread (consumer as well) is able to consume because the buffer has a full slot now (that is why the consumer was awakened). When the awakened thread accesses the monitor, it does not check the condition of the buffer having items to withdraw again because the condition is not in a while loop, so it goes directly to withdraw the item which causes the program to fail because the buffer is empty.

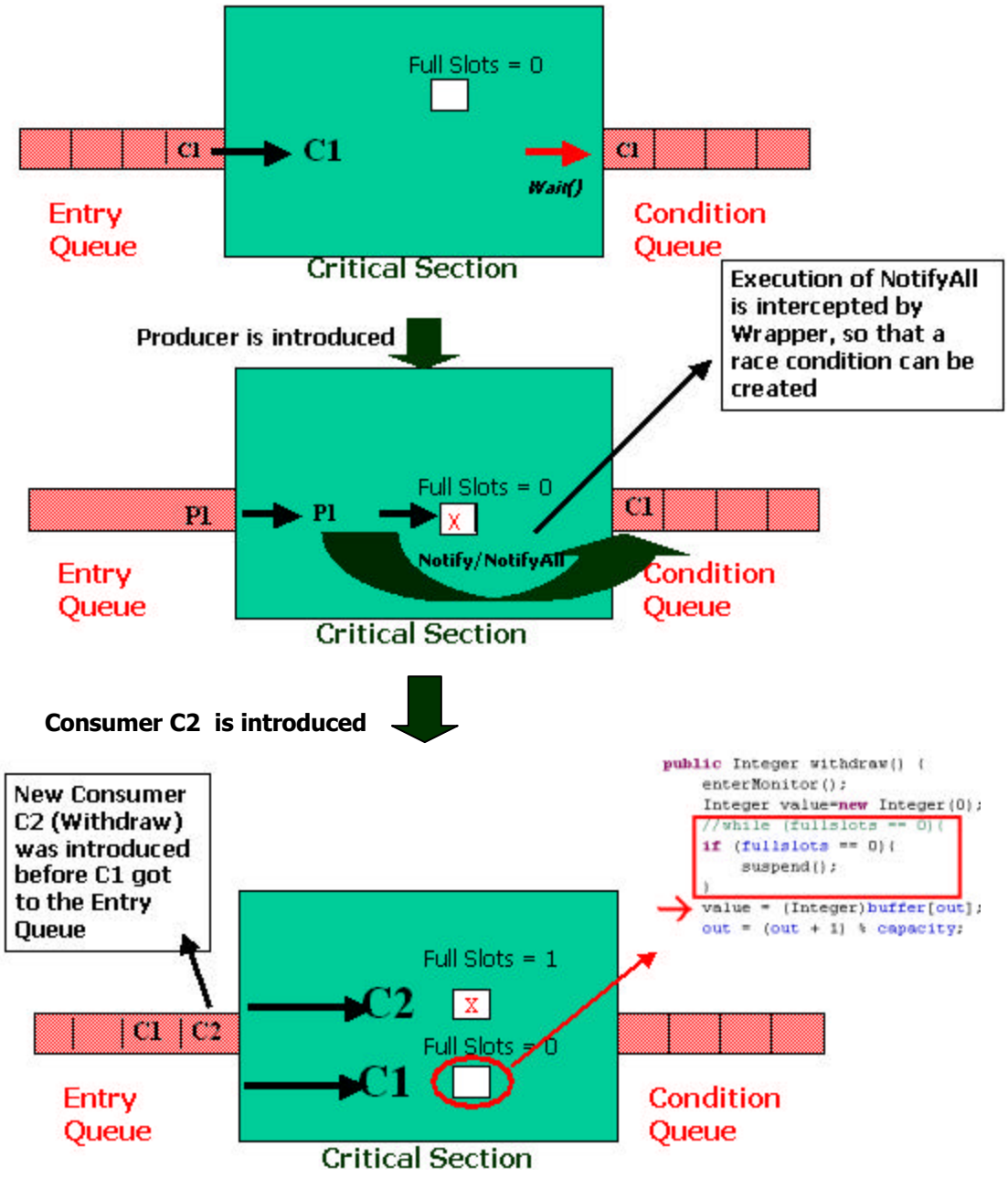

Figure 5.1 Bounded Buffer Experiment Example

5.4 Experiments Results

Table 5.1 Experiments Results

Monitor	# of Requirements	# of Mutants	# of Mutants Killed	# of Paths Explored	# of Transitions Executed	# of States Explored	Exploration Time
Bounded Buffer	7	12	12	15	47	33	3.2 seconds
Safe Bridge	6	10	10	38	94	57	4.859 seconds
Writers Readers	6	12	12	38	98	61	5.687 seconds

Table 5.1 shows that all mutants applied to each monitor were killed by our tool, which confirms that the state space exploration approach is able to effectively detect synchronization faults caused by common programming errors. The exploration time indicated in the table also shows that our tool has a good performance, being able to explore all the state space in less than 6 seconds in all cases.

CHAPTER 6

CONCLUSIONS AND FUTURE ENHANCEMENTS

6.1 Conclusions

This thesis shows that the state exploration approach can be successfully used to do unit testing for concurrent programs, therefore helping developers find problems that would be very hard to find with traditional sequential programs testing approaches or by manually creating test sequences and executing them. The main contribution of this thesis is the on-the-fly introduction of threads to dynamically generate test sequences, using rules that will always try to create race conditions and therefore detect problems when more than one thread tries to access a Java Monitor at the same time. The experiment with the Bounded Buffer, for example, showed that our tool was able to detect problems due to race conditions when the *wait()* operation was in a *if* statement rather than a *while* statement, so notified threads don't check the condition again before entering the critical section. This creates an error condition when other threads barged ahead. This type of problem would be very difficult to detect by a developer without having help from a tool like ours that explores the state space of the monitor and dynamically builds test sequences at the same time they are executed.

The experiments made with three different Monitors, the Bounded Buffer, the Writers and Readers and the Safe Bridge, show that our tool effectively killed all the mutants that were exercised. The mutants were defined based on common

synchronization errors. The experiments also confirmed that using the Abstract State is an effective way to store only the information necessary to explore and evaluate the monitor, and still ensure that the exploration reached a final point. As shown in the results summary, all three monitors took less than six seconds to explore all the java monitor space, and even less than that when problem conditions are found, so the performance of the tool is very reasonable, which is important when a lot of states have to be explored. More testing needs to be done with industrial type monitors to make sure that the performance still holds.

The Execution module along with the Wrapper was able to control the execution of non-deterministic test runs, which not only allows the exploration to decide what paths to build without processing duplicates states, but also enables the user to reply sequences for regression testing. Other approaches described in this thesis realized that the main reason why their tool would not find all the errors that were introduced was because they could not know when a thread was done making a call. This is a significant advantage of our tool, since we are able to control the execution and intercept the thread synchronization operations such as *wait* and *notify/notifyAll*. This way the tool can update the monitor state and make decisions based on that.

My main contribution in this work was the design and implementation of the Driver/Execution module, the Monitor Wrapper that intercepts operations in the Java Monitor and the Evaluation module that provides the APIs and support classes for the detection of error/warning conditions.

6.2 Future Enhancements

Future enhancements to our approach and tool would be the automatic transformation of monitor class from regular java calls to wrapper calls, so that the user does not have to worry about making the file ready for the tool. In addition, a more thorough evaluation of the approach with different monitors is needed in order to refine the tool and identify what other problems can be detected. Likewise, it would be good to compare it to the effectiveness of the other approaches described in the thesis to see how ours compares. A graphical user interface would also be helpful for the developers to visualize the state exploration process. In addition, a protocol that the user can use to specify the order of execution for the monitor methods should be implemented, so that the exploration has more information to determine the enabled transitions. In the Safe Bridge problem, for example, the method eastEnter should be executed before eastExit. The tool currently does not support this, so the code in the methods has to handle this requirement.

6.3 Final Remarks

The main differentiator of the approach and implementation presented in this thesis compared to existent approaches, is that our tool dynamically builds test sequences introducing threads on the fly during exploration time, in order to create race conditions and therefore detect synchronization problems. Unlike other approaches, our tool is able to test the components directly so an additional program does not have to be developed in order to test the monitor. Furthermore, the number of threads needed to execute a test does not need to be known a priori, because of our ability to make

decisions during exploration as needed while we build the test sequences automatically. This is also possible due to the level of control of the execution that we could reach with this approach and implementation. With our state space exploration approach we were able to introduce a considerable level of automation to the testing process that is available today to the best of our knowledge, thus reducing the amount of manual intervention that the user has to perform in order to test a Java Monitor and enabling the detection of different synchronization faults that were not detected before by other approaches.

APPENDIX A

BOUNDED BUFFER EXPERIMENT

APPENDIX A: Bounded Buffer Experiment

1. Requirements

Requirement #	Requirement Description
1	If buffer is full Producer cannot produce
2	If buffer is empty Consumer cannot consume
3	If buffer is not full Producer should be able to produce
4	If buffer is not empty Consumer should be able to consume
5	An item cannot be overridden
6	An item cannot be consumed twice
7	Consumers and Producers should not be waiting at the same time

2. Abstract State

Data Members Abstraction

Data Members	**FullSlots**	**Valid**
0	**FullSlots=0**	**Y**
0-N	**0<FullSlots<N**	**Y**
N	**FullSlots=N**	**Y**
N++	**FullSlots>N**	**N**
0--	**FullSlots<0**	**N**

Entry Queue Abstraction: {withdraw,deposit,""} Each value represents the type of thread in the head of the entry queue, blank if it is empty

Critical Section Abstraction: {withdraw,deposit,""} Each value indicates the type of thread in the Critical Section, blank if there is no thread active in the monitor

Condition Queue Abstraction:

Examples: {withdraw,deposit},{withdraw,deposit+},{"",deposit}

These values indicate the type of threads in the condition queue, and whether there is only 1 thread (i.e. withdraw) or more than one thread (i.e. withdraw+).

3. Source Code

3.1 Correct code after transformation for testing

```
package edu.uta.cse.Monitor;

import edu.uta.cse.MonitorTesting.Execution.MonitorToolBoxWrapper;
import java.lang.Integer;


public class BoundedBuffer extends MonitorToolBoxWrapper {
	public int fullslots=0;
	private int capacity = 0;
	private Integer[] buffer = null;
	private int in = 0, out = 0;

	public BoundedBuffer() {
		this.buffer=new Integer[this.capacity];
	}

	public BoundedBuffer(int capacity, int fullSlots) {
		this.fullslots=fullSlots;
		this.capacity=capacity;
		this.buffer=new Integer[this.capacity];
	}

	public void deposit(Integer value) {
		enterMonitor();
		while(fullslots == capacity) {
			suspend();
		}
```

```
                buffer[in]=value;
                in=(in + 1)%capacity;
                if(this.fullslots++ == 0) {
                        signalAll();
                }
                exitMonitor();
        }

        public Integer withdraw() {
                enterMonitor();
                Integer value=new Integer(0);
                while (fullslots == 0){
                        suspend();
                }
                value = (Integer)buffer[out];
                out = (out + 1) % capacity;
                if (fullslots-- == capacity){
                        signalAll();
                }
                exitMonitor();
                return value;
        }

}
```

4. Code change for experiment with Mutant while => if in *withdraw()* method

4.1 Initialize Function

```
this.boundedBuffer=new BoundedBuffer(10, 0); //capacity,fullSlots
```

4.2 Withdraw method changed with mutant applied

```
public Integer withdraw() {
                enterMonitor();
                Integer value=new Integer(0);
                //while (fullslots == 0){ //CORRECT
                if (fullslots == 0){       //INCORRECT
                        suspend();
                }
                value = (Integer)buffer[out];
                out = (out + 1) % capacity;
                if (fullslots-- == capacity){
```

```
                signalAll();
            }
            exitMonitor();
            return value;
        }
```

5. Program results

```
BOUNDED BUFFER: Please enter the path of input file:
c:\input.txt
Path: 1 begins....
Path: 2 begins....


---- CONDITION FOUND!!! ----
Condition #: 02 Type: ERROR
Category   : lower_bound
Description: Data value is lower than lower bound
Requirement: 02 - If buffer is empty Consumer cannot consume
ATTRIBUTES :
Name       : fullslots current value: -1

PRINTINT STATES STACK IN THIS PATH...

----------------Start of Abstract State----------------
State #: 1
Entry Queue is Empty
Critical Section is Empty
Condition Queue is Empty
The value of data element is 0
----------------Start of Abstract State----------------
State #: 2
Thread at the head of the Entry Queue is withdraw
Critical Section is Empty
Condition Queue is Empty
The value of data element is 0
----------------Start of Abstract State----------------
State #: 3
Entry Queue is Empty
Critical Section is Empty
Thread in the Condition Queue is withdraw
The value of data element is 0
----------------Start of Abstract State----------------
State #: 4
Thread at the head of the Entry Queue is withdraw
Critical Section is Empty
Thread in the Condition Queue is withdraw
The value of data element is 0
----------------Start of Abstract State----------------
State #: 5
Entry Queue is Empty
```

```
Critical Section is Empty
Thread in the Condition Queue is withdraw+
The value of data element is 0
----------------Start of Abstract State---------------
State #: 6
Thread at the head of the Entry Queue is deposit
Critical Section is Empty
Thread in the Condition Queue is withdraw+
The value of data element is 0
----------------Start of Abstract State---------------
State #: 7
Entry Queue is Empty
Thread in Critical Section is deposit
Thread in the Condition Queue is withdraw+
The value of data element is 0-N
----------------Start of Abstract State---------------
State #: 8
Thread at the head of the Entry Queue is withdraw
Thread in Critical Section is deposit
Condition Queue is Empty
The value of data element is 0-N
----------------Start of Abstract State---------------
State #: 9
Thread at the head of the Entry Queue is Rewithdraw
Thread in Critical Section is withdraw
Condition Queue is Empty
The value of data element is 0
----------------Start of Abstract State---------------
State #: 10
Thread at the head of the Entry Queue is Rewithdraw
Thread in Critical Section is withdraw
Condition Queue is Empty
The value of data element is 0--
STATE NOT VALID!!!!!!!!!!
Path: 3 begins....
Path: 4 begins....
Path: 5 begins....
Path: 6 begins....
Path: 7 begins....
Path: 8 begins....
Path: 9 begins....
Exploration has ended...


Total Paths explored: 9
Unique Transitions executed are: 28
Unique States explored: 20
Total Exploration Time (in milliseconds): 890
```

APPENDIX B

SAFE BRIDGE EXPERIMENT

APPENDIX B: Safe Bridge Experiment

1. Requirements

Requirement #	Description
01	Cars coming from different directions cannot access the bridge at the same time
02	If no West cars in the bridge, East cars should be able to access
03	If no East cars in the bridge, West cars should be able to access
04	East or West cars can only exit once they have entered
05	If at least one East car is in the bridge, all east cars should be able to access the bridge
06	If at least one West car is in the bridge, all west cars should be able to access the bridge

2. Abstract State

Data Members Abstraction

Data Members	West Cars	East Cars	Valid
0;0	0	0	Y
0;E	0	>=1	Y
W;0	>=1	0	Y
W;E	>=1	>=1	N
-;-	<0	<0	N

Entry Queue Abstraction: {west,east,""} Each value represents the type of thread in the head of the entry queue, blank if it is empty

Critical Section Abstraction: {west,east,""} Each value indicates the type of thread in the Critical Section, blank if there is no thread active in the monitor

Condition Queue Abstraction:

Examples: {west,east},{west,east+},{"",east}

These values indicate the type of threads in the condition queue, and whether there is only 1 thread (i.e. west) or more than one thread (i.e. west+).

3. Source Code

3.1 Correct code after transformation for testing

```
package edu.uta.cse.Monitor;

/**
* @author Monica Hernandez
* Single lane bridge problem
* The bridge is going in directions East to West (W) and West to East (E)
* so the program has to ensure that no cars going in opposite directions can
* access the bridge at the same time to avoid collisions
* Cars going from East to West are labeled W
* Cars going from West to East are labeled E
*/

import edu.uta.cse.MonitorTesting.Execution.MonitorToolBoxWrapper;

public class SafeBridge extends MonitorToolBoxWrapper{

        public int westCars = 0;
        public int eastCars = 0;

        public SafeBridge(int eastCars, int westCars){
```

```
            this.eastCars = eastCars;
            this.westCars = westCars;
        }

        public void westEnter() throws InterruptedException {
            enterMonitor();
            while (eastCars>0)
                suspend();
            ++westCars;
            exitMonitor();
        }

        public void westExit(){
            enterMonitor();
            if (this.westCars >0){  //execute only if there are W cars in bridge
                --westCars;
                if (westCars==0){
                    signalAll();
                }
            }
            exitMonitor();
        }

        public void eastEnter() throws InterruptedException {
            enterMonitor();
            while (westCars>0)
                suspend();
            ++eastCars;
            exitMonitor();
        }

        public void eastExit(){
            enterMonitor();
            if (this.eastCars >0){ //execute only if there are E cars in bridge
                --eastCars;
                if (eastCars==0){
                    signalAll();
                }
            }
            exitMonitor();
        }
}
```

4. Code change for experiment with mutant notifyAll => notify

4.1 Initialize Function

```
this.safeBridge=new SafeBridge(0,1); //numEast, numWest
```

4.2 *eastExit()* method changed

```
public void eastExit(){
        enterMonitor();
        if (this.eastCars >0){ //execute only if there are E cars in bridge
                --eastCars;
                if (eastCars==0){
                        //signalAll(); //CORRECT
                        signal(); //INCORRECT
                }
        }
        exitMonitor();
}
```

5. Program results

```
SAFE BRIDGE TEST: Please enter the path of input file:
c:\input-bridge.txt
Path: 1 begins....
Path: 2 begins....
Path: 3 begins....
Path: 4 begins....


---- CONDITION FOUND!!! ----
Condition #: 03 Type: ERROR
Category   : starvation
Description: West Cars waiting when there are no east cars in the
bridge
Requirement: 03 - If no East cars in the bridge, West cars should be
able to access
ATTRIBUTES :
Name       : westCars current value: 0
Name       : eastCars current value: 0

PRINTINT STATES STACK IN THIS PATH...

---------------Start of Abstract State---------------
```

```
State #: 1
Entry Queue is Empty
Critical Section is Empty
Condition Queue is Empty
The value of data element is 0;E
----------------Start of Abstract State----------------
State #: 2
Thread at the head of the Entry Queue is westEnter
Critical Section is Empty
Condition Queue is Empty
The value of data element is 0;E
----------------Start of Abstract State----------------
State #: 3
Entry Queue is Empty
Critical Section is Empty
Thread in the Condition Queue is westEnter
The value of data element is 0;E
----------------Start of Abstract State----------------
State #: 4
Thread at the head of the Entry Queue is westEnter
Critical Section is Empty
Thread in the Condition Queue is westEnter
The value of data element is 0;E
----------------Start of Abstract State----------------
State #: 5
Entry Queue is Empty
Critical Section is Empty
Thread in the Condition Queue is westEnter+
The value of data element is 0;E
----------------Start of Abstract State----------------
State #: 6
Thread at the head of the Entry Queue is eastExit
Critical Section is Empty
Thread in the Condition Queue is westEnter+
The value of data element is 0;E
----------------Start of Abstract State----------------
State #: 7
Entry Queue is Empty
Thread in Critical Section is eastExit
Thread in the Condition Queue is westEnter+
The value of data element is 0;0
----------------Start of Abstract State----------------
State #: 8
Thread at the head of the Entry Queue is westEnter
Thread in Critical Section is eastExit
Thread in the Condition Queue is westEnter
The value of data element is 0;0
STATE NOT VALID!!!!!!!!!!
Path: 5 begins....
```

APPENDIX C

WRITERS AND READERS EXPERIMENT

APPENDIX B: Writers and Readers Experiment

1. Requirements

Requirement #	Description
01	Readers and writers should not access the shared variable at the same time
02	Only one writer should access the variable at the same time
03	If no readers reading the variable, one writer should be able to access the variable
04	If no writers writing the variable, all readers should be able to access the variable
05	If at least one reader is reading, all the other readers should be able to read the variable

2. Abstract State

Data Members Abstraction

Data Members	Writers	Readers	Valid
0;0	0	0	Y
0;R	0	1	Y
W;0	1	0	Y
0;R+	0	>1	Y
W;R	1	1	N
W;R+	1	>1	N
W+;0	>1	0	N

Entry Queue Abstraction: {writer,reader,""} Each value represents the type of thread in the head of the entry queue, blank if it is empty

Critical Section Abstraction: {writer,reader,""} Each value indicates the type of thread in the Critical Section, blank if there is no thread active in the monitor

Condition Queue Abstraction:

Examples: {writer,reader},{writer,reader+},{“”,reader}

These values indicate the type of threads in the condition queue, and whether there is only 1 thread (i.e. reader) or more than one thread (i.e. reader+).

3. Source Code

3.2 Correct code after transformation for testing

```
package edu.uta.cse.Monitor;

import edu.uta.cse.MonitorTesting.Execution.MonitorToolBoxWrapper;

/**
 *@author: j.n.magee 11/12/96
 *@author: Monica Hernandez 11/01/05 - modified to use wrapper methods
 */

/**
* A solution to the Readers/Writers
*problem. Multiple readers can access a shared variable at the
*same time, whereas a writer must obtain mutually exclusive
*access. In this solution, a writer may starve, i.e. a writer may
*never get a chance to access the variable
*/
public class ReaderWriterSafe extends MonitorToolBoxWrapper  {

	public int numReaders =0;
	public int numWriters = 0;

	public ReaderWriterSafe(int readers, int writers) {
		super();
		this.numReaders = readers;
		this.numWriters = writers;
	}

	public void read() throws InterruptedException {
		enterMonitor();
		while (numWriters>0){
			suspend();
		}
		++numReaders;
```

```
            exitMonitor();
        }

        public void releaseRead() {
            enterMonitor();
            if (this.numReaders > 0){ //release read only if there are readers reading
                --numReaders;
                if(numReaders<0){
                    signal();
                }
            }
            exitMonitor();
        }

        public void write() throws InterruptedException {
            enterMonitor();
            while (numReaders>0 || numWriters>0){
                suspend();
            }
            ++numWriters;
            exitMonitor();
        }

        public void releaseWrite() {
            enterMonitor();
            if (this.numWriters>0){ //release write only if there are writers writing
                --numWriters;
                signalAll();
            }
            exitMonitor();
        }
}
```

4. Code change for experiment with mutant '=' => '>' in method *releaseRead()*

4.1 Initialize Function

```
this.monitorObject = new ReaderWriterSafe(1,0); //reader, writer
```

4.2 *releaseRead()* method changed

```
public void releaseRead() {
            enterMonitor();
```

```
            if (this.numReaders > 0){ //release read only if there are readers reading
                --numReaders;
                //if(numReaders==0){  //CORRECT
                if(numReaders<0){     //INCORRECT
                    signal();
                }
            }
            exitMonitor();
        }
```

5. Program results

```
WRITERS READERS SAFE TEST: Please enter the path of input file:
c:\input-wr.txt
Path: 1 begins....
Path: 2 begins....

---- CONDITION FOUND!!! ----
Condition #: 03 Type: ERROR
Category   : possible_starvation
Description: Writers waiting to write when there are no readers
reading the variable
Requirement: 03 - If no readers reading the variable, one waiting
writer should be able to access the variable

Data Elements:
Name: numWriters current value: 0
Name: numReaders current value: 0

PRINTINT STATES STACK IN THIS PATH...

----------------Start of Abstract State----------------
State #: 1
Entry Queue is Empty
Critical Section is Empty
Condition Queue is Empty
The value of data element is 0;R
----------------Start of Abstract State----------------
State #: 2
Thread at the head of the Entry Queue is write
Critical Section is Empty
Condition Queue is Empty
The value of data element is 0;R
----------------Start of Abstract State----------------
State #: 3
Entry Queue is Empty
Critical Section is Empty
Thread in the Condition Queue is write
The value of data element is 0;R
----------------Start of Abstract State----------------
State #: 4
```

```
Thread at the head of the Entry Queue is write
Critical Section is Empty
Thread in the Condition Queue is write
The value of data element is 0;R
----------------Start of Abstract State----------------
State #: 5
Entry Queue is Empty
Critical Section is Empty
Thread in the Condition Queue is write+
The value of data element is 0;R
----------------Start of Abstract State----------------
State #: 6
Thread at the head of the Entry Queue is releaseRead
Critical Section is Empty
Thread in the Condition Queue is write+
The value of data element is 0;R
----------------Start of Abstract State----------------
State #: 7
Entry Queue is Empty
Critical Section is Empty
Thread in the Condition Queue is write+
The value of data element is 0;0
STATE NOT VALID!!!!!!!!!!
Path: 3 begins....
```

REFERENCES

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

[2] C. Harvey and P. Strooper (2001), Testing Java monitors through deterministic execution, Proc. of Australian Software. Engineering Conference, pp. 61-67.

[3] J. Magee and J. Kramer (1999), Concurrency: State Models & Java Programs, John Wiley & Sons, Chichester, England.

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

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

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

[7] Richard H. Carver and Kuo-Chung Tai (1991). Replay and Testing for Concurrent Programs, IEEE Software, pp. 66-74.

[8]http://www.cise.ufl.edu/research/ParallelPatterns/PatternLanguage/Background/Glossary.htm#R

BIOGRAPHICAL INFORMATION

Monica Hernandez received her B.S. in Systems Engineering from EAFIT University, Medellin-Colombia, in June 2001. Her majors were Databases and Software Engineering, and her thesis was "Hermes: A web based computer assisted data retrieval and analysis tool". She received her M.S. degree in Computer Science and Engineering from University of Texas at Arlington in December 2005. Her research interests include information retrieval in web based applications, and testing of object oriented and concurrent programs.