MONITOREXPLORER: A STATE-SPACE EXPLORATION BASED TOOL TO TEST JAVA MONITORS IMPLEMENTATIONS

by

VIDUR GUPTA

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

THE UNIVERSITY OF TEXAS AT ARLINGTON

May 2006

ACKNOWLEDGEMENTS

I would like to thank Dr. Jeff Lei, my supervisor and teacher. He has guided me not only through my thesis but also with various aspects of my student and professional life. I would also like to mention Monica and Chinmay who also worked on this project. The discussions and deliberations we had were simulating and educating. I would also like to thank other members of the Software Engineering Research Group led by Dr. Lei and my fellow students at UTA. These people have always been there to help and support me during my stay at Arlington.

I must acknowledge the support and assistance of the staff of the Computer Science and Engineering Department. I would specially like to thank Ms. Costabile, Graduate Program Secretary and Ms. McBride, CSE Department Office Manager.

Another group worth mentioning would be my friends from India who have given me hope and encouragement whenever things got difficult.

No acknowledgements would be complete without the mention of my family who gave me the opportunity to come and study in this institution. They have given me every support and strength. I can never thank them enough.

April 13, 2006

ABSTRACT

MONITOREXPLORER: A STATE-SPACE EXPLORATION
BASED TOOL TO TEST JAVA MONITORS
IMPLEMENTATIONS

Publication No. _____

Vidur Gupta, M.S.

The University of Texas at Arlington, 2006

Supervising Professor: Dr. Jeff Lei

A monitor is a concurrency construct that encapsulates data and functionality for allocating and releasing shared resources [1]. Java associates a lock with every object. When a block of code is guarded by the 'synchronized' keyword then the thread has to get lock on objects inside the block. This block is a Java Monitor which guarantees mutual exclusion and the thread needs to get the locks before it can enter the monitor. The locks are released when the thread exits the guarded block. If the thread is not able to get all the locks it waits for the conditions to be right.

There are many application classes which are written using the Java Monitors and these are difficult to test due to the inherent complexities of the concurrent programs. The key challenge is to be able to trace all possible execution paths and then able to reproduce them for regression testing. Our work explores the state-space of the monitor application. The state space is explored in the depth first fashion. At each state the possible transitions are calculated pushed on a stack. The transition on the top of the stack is executed. This process is repeated till a duplicate or invalid state is detected.

The key features of the approach are data abstraction which helps reduce the number of states to be explored and introduction of threads on the fly to execute the selected transition that simulate the race conditions. We have developed 'MonitorExplorer', a tool that can used to test monitor applications. This tool has been used to test various applications and their mutants. The experimental results show that the tool is effective in detecting synchronization bugs.

TABLE OF CONTENTS

LIST OF ILLUSTRATIONS

LIST OF TABLES

Table Page

CHAPTER 1

INTRODUCTION

Traditionally multithreaded programs were used in operating systems but now many application programs are also multithreaded. Some examples of multithreaded applications used commonly are Web Servers, Database Management Systems, PC Games and many more. The key issue is access to shared resources between various threads. The problem becomes worrisome when more than one of the threads wants to write in the shared space. Another issue is that of non-determinism, which is due to the fact that the programmer has no control on when and how long will a thread be assigned the CPU time. This makes it very difficult to reliably test and reproduce an error during debugging. A multithreaded program which is executed with the same inputs a number of times might yield different results on each trial. This problem does not exist in a sequential program. The following example illustrates this: -

Assume three data elements, x, y and z, initialized to 0.

Thread 1: x = y + z;
y = 1;
z = 2;

The possible result is: - x = 0.

Figure 1.1 Test result for sequential program

The following example illustrates the non-determinism in a very simple application which adds values of two variables and stores them in a third variable [2]: -

Assume three data elements, x, y and z, initialized to 0.

```
Thread 1: x = y + z;
Thread 2: y = 1;
          z = 2;
```

The above high language constructs are converted into the following machine code: -

```
Thread 1:
        load r1, y;   -   (1)
        add r1, z;    -   (2)
        store r1, x;  -   (3)
Thread 2:
        assign y, 1; -   (4)
        assign z, 2; -   (5)
```

The possible interleaving are: -

- (1), (2), (3), (4), (5) and x = 0.
- (1), (2), (4), (5), (3) and x = 0.
- (1), (2), (4), (3), (5) and x = 0.
- (1), (4), (5), (2), (3) and x = 2.
- (4), (1), (2), (3), (5) and x = 1.
- (4), (5), (1), (2), (3) and x = 3.

Figure 1.2 Possible interleaving and results for multithreaded program

Our work focuses on testing Monitors which are a special synchronization construct. The testing of monitors is not trivial because of the interleaving. Also, it difficult to determine how many threads we need to reliably test the monitor.

We now take a look at a generic monitor and then at a Java Monitor. Also, we would see how these two are different and what makes testing them a difficult problem.

1.1 Monitor

Monitor is a high-level synchronization construct that supports data encapsulation and information hiding. It encapsulates shared data, operations on data and the required synchronization for accessing the data [2]. Monitors guarantees mutual exclusion. Figure 1.3 illustrates the run-time structure of a monitor.

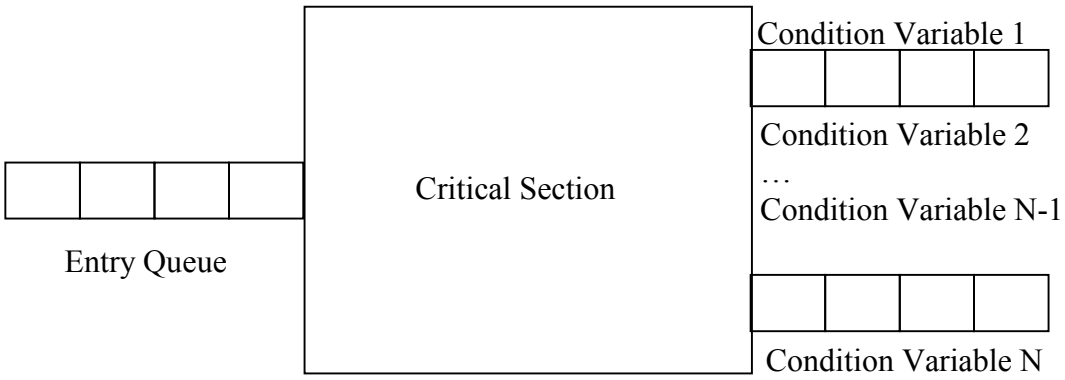

Figure 1.3 Graphic view of a monitor

A monitor has the following parts: -

- Critical Section: - This section guards the shared data elements. A thread needs to enter this section to be able to access the shared data element and execute synchronized operations. There can only be one thread in the critical section at any point of time.
- Entry Queue: - All threads wait here for the entry into the critical section. This is a FIFO queue and the thread at the head of queue is allowed to enter the critical section when it is empty.

- Condition Queues: - Each condition queue has a condition variable. A thread waiting for a condition variable to be true waits in the related condition queue. Traditionally condition queues are FIFO in nature.

A thread enters the condition queue when it executes the wait() operation on the related conditional variable. It is either placed back in the entry queue or in a special re-entry queue depending on the signaling discipline when the signal() method on the condition variable is executed by the thread in the critical section. We will discuss one of the signaling disciplines in the next section.

1.2 Java Monitor

A Java Monitor is a Java class which provides synchronized data access. The synchronization is implemented using the keyword 'synchronized' to classify a method. It guarantees mutual exclusion with respect to the access to the shared data elements.

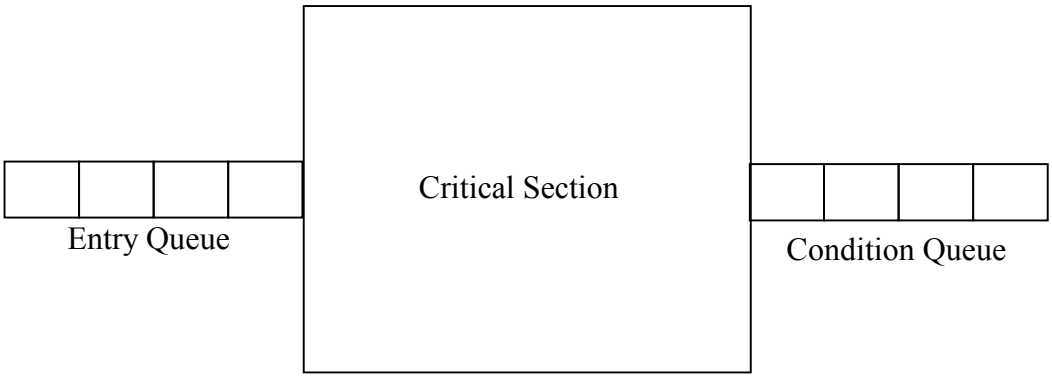

Figure 1.4 Graphic view of a Java monitor

Figure 1.4 shows the graphical view of a Java Monitor at run-time. One noticeable difference between the structure of a Java Monitor and a general monitor is that in the general monitor there a condition queue associated with each condition

variable whereas in the Java Monitor there is only one such queue for all condition variables. A Java monitor has the following parts: -

- Critical Section: - The critical section in a Java Monitor is similar to the critical section of a traditional monitor. It guards the shared data elements. A thread needs to enter this section to be able to access the shared data element and execute synchronized operations. There can only be one thread in the critical section at any point of time.
- Entry Queue: - Any thread which executes the 'synchronized' method waits in the entry queue. This queue is a FIFO queue like the traditional monitor.
- Condition Queue: - This where the Java monitor differs from the traditional monitor. The first difference is that unlike the traditional monitor the Java monitor just has one queue where all threads wait no matter for which condition variable they are waiting for. The second difference is that this queue is not FIFO in nature. It is random for the signal() operation. Thus when a signal() operation is exerted then any thread in the condition queue is awakened. This also means that a thread which may be waiting for some other condition to be true might be awakened.

The condition queue's property requires that we have a mechanism which allows us to ensure that we do not end up in a situation where a thread is still waiting in the condition queue even when the condition is right for it to enter the critical section. Java has added the signalAll() method for this. It awakens all the threads in the

condition queue and puts them back in the entry queue. The signaling discipline explained will illustrate how this works.

Another important aspect about monitors is the signaling discipline. This defines how a thread when in the condition queue will behave when awakened and also how the thread in the critical section can behave after executing a signal() operation. The signaling discipline for a Java monitor is the Signal-and-Continue. This means that the thread in the critical section can execute the signal() operation and then continue its work in the critical section. The signaled thread goes to the end of the entry queue and is treated like any other thread in the entry queue.

Now we know the runtime structure and the behavior of Java Monitors, next we look in to what makes testing of such programs more challenging than traditional testing techniques.

1.3 Java Monitor Testing

Testing a sequential program or a program with a single thread of execution is relatively easier as we saw in Figure 1 and Figure 2. This is because multiple threads are executing at any point of time in a concurrent system. As we have seen there can only be one thread in the critical section and multiple threads waiting in the entry queue and the condition queue. Thus, in order to reliably test the Java Monitor we need to create multiple threads. This leads to another problem. We have little control over neither the operating system's scheduler nor the Java Runtime. Thus we cannot predictably execute threads in a given order. Thus we cannot ensure a degree of

satisfaction i.e. we cannot say that we have tested 99% of possible paths, etc. The major issues are: -

- Non-deterministic behavior: - As discussed earlier this is an issue with all concurrent programs. The challenge is how to reliably produce and reproduce a behavior during testing and regression.
- Number of Threads: - Threads are an operating system resource. They consume memory, CPU cycles and other system resources. Thus it would not be prudent or possible to have unlimited number of threads. On the other hand we have seen that one thread is not enough. So, the challenge is how many threads we need.
- Runtime Information of Java Monitor: - Java does not provide any control over the Java Monitor. It also provides no information about the number of threads in the queues or the thread in the critical section at run-time. Thus we need a mechanism which simulates the behavior of the Java Monitor yet gives us all this information.

We have addressed the key challenges as listed above in our work. As we will illustrate, we are able to exercise well defined possible states the monitor can reach by exploring the state-space in a depth-first fashion. Chapter 2 discusses the MonitorTest Algorithm which explores the state-space. We also exercise all possible transitions that can be executed at each monitor state.

We have replicated the behavior of the Java Monitor using our Monitor Toolbox [3]. It simulated the FIFO nature of the entry queue, guarantees mutual exclusion for the critical section and provides for random waking of threads on the signal() operation.

We also introduce threads on the fly and maintain a Thread Pool thus reducing the overhead of creating a thread each time we need one. This pool adjusts in size based on the need of threads thus reducing the load on the system. Chapter 3 would discuss the implementation of the MonitorExplorer tool and the key packages.

We introduce the concept of abstraction later in the document. It allows us to focus on key states and ignore states where the monitor would exhibit similar behavior. The trade-off is between the level of abstraction and the size of the state-space to be explored. The finer the abstraction the more the state-space would be. This would increase the reliability of the tests but increase the computation time. On the other hand if the abstraction is too coarse we might miss out on some important states and transitions. This would however a small state-space thus a less execution time.

We have conducted several experiments to show the effectiveness of our approach. We have used some of the classical monitor implementation and introduced the common synchronization errors. The effectiveness of the tool is measured by the percentage of mutants killed by the MonitorExplorer tool. We also track the time of execution to show that the tests are conducted in reasonable time such that it can be used in real-life scenarios. In Chapter 4 we present our experiments and results.

Chapter 5 discusses related work. In Chapter 6 we present our conclusion and in Chapter 7 share the future work.

My main contribution to the MonitorExplorer is the MonitorTesting module. It includes the Controller, ThreadPool and Utilities sub-packages. I implemented the MonitorTest algorithm and the getEnabledTransitions() method. I also implemented the logic for retrace, interaction between the user & the system and designed the data structures like Transition and AbstractState. Another important component that I designed and implemented was the communcationMonitor() which enforces synchronization between the main thread and executing threads.

CHAPTER 2

MONITORTEST: THE ALOGRITHM

The MonitorTest Algorithm is a depth-first algorithm which traverses the state-space of the Java Monitor. The algorithm is illustrated in Figure 2.1. As illustrated in the algorithm there are two key data structures that we maintain. One is the set of visited states. Thus data structure is used to keep track of the states we have visited. It usefulness will be illustrated shortly. The other data structure is called stack which is a stack of all possible transitions that can be executed. This stack is populated by the getTransitions() method.

An important thing to mention here is that we do not store the actual monitor state but we store the abstract state which is derived from the actual state. This is a very powerful tool which allows us to reduce the state-space size. Another is that initial state of the monitor can be specified the user and we can start working from there. This allows the tester to exercise a more restrictive path he wants to explore while regression.

In this chapter we would discuss the key components of the algorithm and then describe the flow of events.

```
Initialize:
1. let stack be an empty stack
2. let visited be an empty set
3. create a instance m of M, and initialize m to state s0
MonitorTest () {
4.          AbstractState state = getAbstractState ();
5.          add state into visited;
6.           transitions = getEnabledTransitions ();
7.          Explore (transitions);
}
Explore (transitions: a set of transitions) {
8.          for (each transition t in transitions) {
9.            push t onto stack
10.           execute t;
11.           state = getAbstractState ();
12.           if (state is valid or state is not in visited) {
13.               add state into visited;
14.               Explore (getEnabledTransitions ());
              }
15.           pop t out of stack;
16.           undo t;
            }
}
```

Figure 2.1 MonitorTest Algorithm

2.1 Abstract State

We would let you consider the following before we go on to discuss the concept and power of abstract state. A monitor with two synchronized methods and one data element with ten possible values can have at least three hundred and twenty possible actual states. The math being that thread waits to execute each of the method and waits the head of the entry queue. Now each of these methods will enter the critical section making it four possible combinations with one thread in the critical section while no thread waits in the entry queue, a thread waits to execute one of the methods or two

threads wait to execute each of the method. Now, multiplying four by the ten possible values for the data element we get forty. And then the same situation in the condition queue as there was in the entry queue makes a total of 320. We still have not considered possibility of having more than one thread with same method in the entry queue or the condition queue.

This problem of exploding state-space increases the computation time required to test. Also, it exponentially increases the number of states and paths of explored thus required lots of memory space. Thus the challenge is to reduce the state-space such that all the key features of the state space are captured and at the same time we keep the use of the resources to the minimum. Also, we need to use this information to determine our next possible transitions based on the state, check for duplicate and invalid states. Our technique of state abstraction addresses exactly this issue.

We would like to add a caveat here that abstraction is a powerful tool with a degree of give-n-take. The finer the granularity of abstraction the more reliable are the results but more time and resource are needed and converse is also true.

As we have seen in the above example the key characteristics of the Java Monitor that we need to capture are: -

- State of thread in the critical section.
- The states of threads in the entry queue.
- The states of threads in the condition queue.
- The value of key data members

Now, to continue with our discussion on the abstract states for the Bounded Buffer (section 2.3) problem we would want to trap enough information so that we know what all transitions can be executed from the current state. To do this we must know what thread is at the head of the entry queue if any. This is the only significant thread in the entry queue as the rest cannot do much till this thread enters the critical section and then gives it up. Also, we really do not care about the thread id, all we really want to know which of the synchronized method this thread will execute. We might also want to know if this is the threads first attempt to enter the critical section or not. Similarly for the critical section and the condition queue what matters to us what methods these threads execute.

The really interesting one is the data element. Let's look at the Bounded Buffer example for clues. The element 'capacity' will remain constant over the execution so if know its value once we do not have to keep track of it. The 'buffer', 'in' and 'out' really do not affect the synchronization behavior which is of interest to us. So, we have narrowed down to 'fullSlots'.

Even for fullSlots not all values are significant to us. More specifically threads will exhibit similar behavior at a group of values. Let 'capacity' be ten, then from value 1 to 9 the Bounded Buffer must have the same behavior. It must allow a producer to deposit and a consumer to consume. However when value is zero it must not allow consumption. Similarly when value is ten it must not allow production to continue. We also must ensure that value of 'fullSlots' does not go beyond ten and below zero. Thus we need to keep track of five types of values for only one data element.

The key elements for the queues that we need to track with the following: -

- Entry Queue: - The method executed by the thread at front of the entry queue or empty if no thread in the entry queue.
- Critical Section: - The method executed by the thread in the critical section or empty if no thread in the critical section.
- Condition Queue: - One entry for each synchronized method if a thread executing that method is present in the condition queue or empty if no thread in the condition queue.

2.2 Enabled Transitions

The second component that is required to explore the state-space is the set of enabled transitions. Let us first see what we mean by a transition. Simply it is a set of actions that must be executed atomically to move from state to another. The keyword is 'enabled'. It means that of all the transitions which when can be executed from a certain state.

The set of enable transitions is built by the getEnabledTransitions() method which pushes them onto a stack. The Explorer() method requests for the method to add new transitions to the stack each time it gets a new valid state. It uses the transition at the top of the stack to proceed. Figure 2.2 illustrates the getEnabledTransitions() algorithm.

```
Set getEnabledTransitions () {
1.      let transitions be an empty set
2.      if (entry queue is not empty and next operation of thread in the CS is
        exit){
3.        create an exit transition and let the thread at the entry queue entry
        }
4.      else if (no thread in entry queue or critical section){
5.        create an Introduce transition for each synchronized method
        }
6.      else if (notify/notifyAll operation with thread in entry queue
        but no thread in condition queue){
7.        create an transition for the thread to continue
        }
8.      else if (notify operation) {
9.        create an Introduce transition for each synchronized method
          and notify for each synchronized method in the condition queue.
        }
10.     else if (notifyAll operation) {
11.       create an Introduce transition for each synchronized method
        }
12.     add these transitions into transitions
13.     return transitions;
}
```

Figure 2.2 getEnabledTransitions() Algorithm

The following is the set of transitions pushed on the stack by the getEnabledTransitions() in the given scenarios: -

- When entry queue is not empty and the thread in CS is about to exit: - Only one transition is created. The next operation is exit executed by the thread in the CS which allows the thread at the head of entry queue.
- When no thread in the entry queue and (the critical section or thread in CS about to exit): - An introduce transition is created for each synchronized

method. Also, if the thread in critical section the exit operation is clubbed with each introduce. In this case introduce is executed before the exit.

- When notify/notifyAll operation but no thread in the condition queue: - One transition is added which tells the thread in the critical section to continue.
- When notifyAll with threads in condition queue: - A transition is added with notifyAll transition and no other action. Then a set of transitions is added with introduce operation for each synchronized method before the notifyAll operation.
- When notify with threads in condition queue: - A set of transitions is added with notify transition for each of the synchronized method in the condition queue and no other action. Then a set of transitions is added with introduce operation for each synchronized method before the each of the notify operations.

The last two sets simulate the race condition where a new thread might barge ahead of the threads awakened from the condition queue. The last set simulates the randomness of the Java Monitor where it may wake up any of the threads waiting in the condition queue on the notify operation. We would like to clarify that the introduce operation means that the thread enter the entry queue. It then waits for it to get to the head of the entry queue and for the critical section to get free before it can enter.

2.3 Bounded Buffer Problem

The example of the classic Bounded Buffer problem will be used to explain the approach and implementation throughout the document. As shown in figure 2.2 there

are two synchronized methods in this problem namely, deposit() and withdraw(). A consumer should be able to withdraw when slots are not empty and a producer should be able to deposit when the slots are not full.

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

Figure 2.3 Bounded Buffer Monitor

Now that we have the bounded buffer monitor let is examine the state abstraction for this monitor.

We have already seen in the previous section that we would really like to just capture the values for fullSlots only. To summarize that discussion the abstract values for fullSlots will be: -

- 0 when fullSlots == 0.
- 0 – N when 0 < fullSlots < capacity
- N when fullSlots == capacity
- 0-- when fullSlots < 0
- N++ when capacity < fullSlots

The last two values would represent invalid state. We would consider the possible states for the entry queue. As discussed earlier we want only to capture the type of threads at the head of the entry queue, thus four possible abstractions are: -

- deposit
- withdraw
- Redeposit
- Rewithdraw

The last two values represent threads which have been to the condition queue and are making an attempt after being signaled. In case of the critical section we again want to capture the type of thread in the critical section, thus possible abstractions are: -

- deposit
- withdraw

The final component in the abstract state is the state of the condition queue. This would be a set with size equal to the number of synchronized methods. Another issue we need to take care of is that there may be some distinction between one thread of a type waiting and multiple threads of the same type. Thus we have the following abstractions: -

- When no thread in condition queue then (“”,””)
- When deposit in condition queue only (“deposit”, “”)
- When withdraw in condition queue only (“”, “withdraw”)
- When one deposit and one withdraw in condition queue (“deposit”, “withdraw”)
- When more than one deposit in condition queue only (“deposit++”, “”)
- When more than one withdraw in condition queue only (“”, “withdraw++”)
- When more than one deposit and more than one withdraw (“deposit++”, “withdraw++”)
- (“deposit++”, “withdraw”)
- (“deposit”, “withdraw++”)

Thus we have 360 possible states. All of these are not valid states and thus the exploration may yield a much smaller state-space. If abstraction is not used the number of possible states would increase exponentially. A simple example would be that we would detect all the eleven valid values of fullSlots and unknown number of illegal values of fullSlots. This alone would increase the number of states of a factor 2.5. If we

go on to consider the numerous possible combinations of thread in the entry queue and condition queue the number of states will explode.

CHAPTER 3

MONITOREXPLORER: THE IMPLEMENTATION

In this chapter we will take a look at the implementation of the MonitorExplorer tool. We would discuss the package structure and the key features of each package. We would also see the flow of control as the tool tests a package.

3.1 Packages

There are four major packages our implementation. Figure 3.1 shows the packages implemented and their interaction.

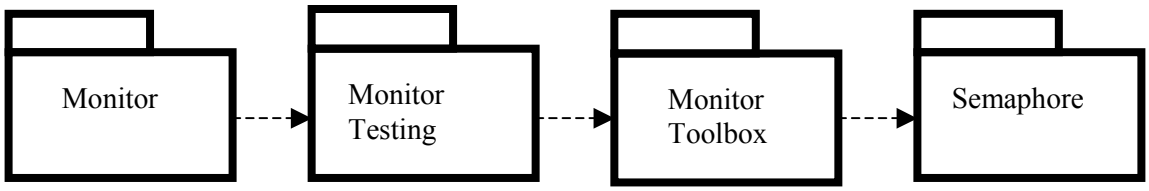

Figure 3.1 Package Diagram

The Monitor package belongs to the user. This is where she defines the monitor to be tested, the abstraction and testing condition. The major components of this package are: -

- Java Monitor: - This is the Java Monitor to be tested. We have to do some instrumentation so that it can be tested. At the beginning of each synchronized method we add enterMonitor() and at the end we have the exitMonitor() call. Also, wait() operation is replaced by suspend(), notify() by signal() and notifyAll() by signalAll().

- UserExtension: - This is where the user defines the abstraction and evaluation conditions. We have provided user APIs which she can use to get the values for the state. Thus the user does not have to worry about the internal structures.

The second component is the brain and brawn of the tool. Here we have the algorithm which builds the transitions and directs the exploration process and the execution module which manipulates the actual monitor, threads & executes the transitions. The major components of this package are: -

- Controller: - Controller is the brain of the tool. It itself is a package with several components. The main part is Controller.java which has the implementation of the algorithm and as the name suggest controls the progress of the tool. It also does the book-keeping and initiates file I/O. It is also the access point for all user entities to interact with lower modules. The other key parts are: -
    - CommunicationMonitor: - This construct is used to synchronize operations between the tool thread and the threads performing operations on the Java Monitor.
    - Data Structures: - The abstract state and transition classes are also defined in this package.
- Execution: - This is brawn of the tool. This package is the execution environment and performs the requests of the controller. The main components of this package are: -

- MonitorDriver: - This is go-through between the controller and the monitor toolbox. We have used the Reflection API provided by Java to extract runtime information of the Java Monitor. Thus we can use generic threads to execute the synchronized methods. This allows user to use the tool without any knowledge of the internal working.
- MonitorToolBoxWrapper: - This is where we have our monitor operation defined. This part does book-keeping at the execution level and builds the information about the actual state of the monitor.
- RunTimeHelper: - This class interacts with the threads executing the synchronized methods.

- Evaluation: - This package provides the user API and the utilities to test the monitor. We would discuss this in greater detail later.
- ThreadPool: - This is a pool of generic threads which adjusts in size as the demand varies. This allows us to reuse threads and reduce the load on the system.
- Utilities: - This package has the file I/O files, commonly used definitions and other utilities.

The third package is the MonitorToolBox. The need for this arises because the Java does not provide any insight into the data structures of its monitor. The data values can be retrieved using Java Reflection API but we also need information like the size and contents of the entry and condition queues. In addition to this we need to know which thread is executing inside the critical section and which method it is executing.

Thus we have tweaked the implementation of a generic monitor to behave like the internal java monitor. It implements the Signal and Continue signaling discipline. It implements the FIFO nature of entry queue. We have not simulated the randomness of the signal() operation. That is simulated by the controller; however we can specify which thread we want to signal. We also do the book-keeping regarding the queues and critical section in this package.

The last package is the Semaphore package which has the semaphore implementations used by the MonitorToolBox to enforce synchronization.

3.2 Evaluation

We earlier talked about the Evaluation module which is a part of the MonitorTesting package. This sub-package is an important component as this is where the checking of the Java Monitor is done. The current and previous states are verified against the conditions provided by the user. The conditions have to be specified by the user as each monitor has a different behavior. We have provided user API which hides the implementation details from the user and allows her to easily define the requirements. The main parts of this module are: -

- Requirement: - In this class we define the requirement code, the description of the requirement and the category like functional or starvation.
- Condition: - Condition is associated with the requirement code, a condition code, the description of the condition. The most important field is the condition type which can be WARNING or ERROR. In case of an error we terminate the execution the path where as in case for warning we continue

the execution. This is useful as in some cases the user may believe the current condition may lead to an error but currently we can proceed to explore further.

- userAPI: We have mentioned this class a number of times. It provides easy access to the monitor state and transition information to the user without knowing the internal functioning. Here are a few examples are: -
    - public boolean threadTypeInConditionQueue(String threadType, AbstractState state): - This method allows the user to check if a thread type is in the condition queue or not.
    - public int getAttributeValue(int attributeNumber): - The user can get the actual value of a shared variable.
- EvaluationRun: - This class stores all the errors and warning over an execution. The MonitorExplorer terminates the current path of execution when error is found and continues to explore the other paths. This class stores conditions over all paths.
- MonitorEvaluation: - An abstract class provides abstract method which define how to associate requirements with conditions and evaluateMonitor() which takes the stack of visited states and stack of transitions on the path. These can be used by the user API for evaluation.

Another class which needs to be mentioned is the UserEvaluation. It is a part of UserExtension sub-package in the Monitor package. This is where the user defines the condition and requirements.

3.3 Sequence

Now that we are familiar with the packages and the static details of the Explorer, let us take a look at the run-time scenario. Figure 3.2 is the sequence diagram for execution of one transition.

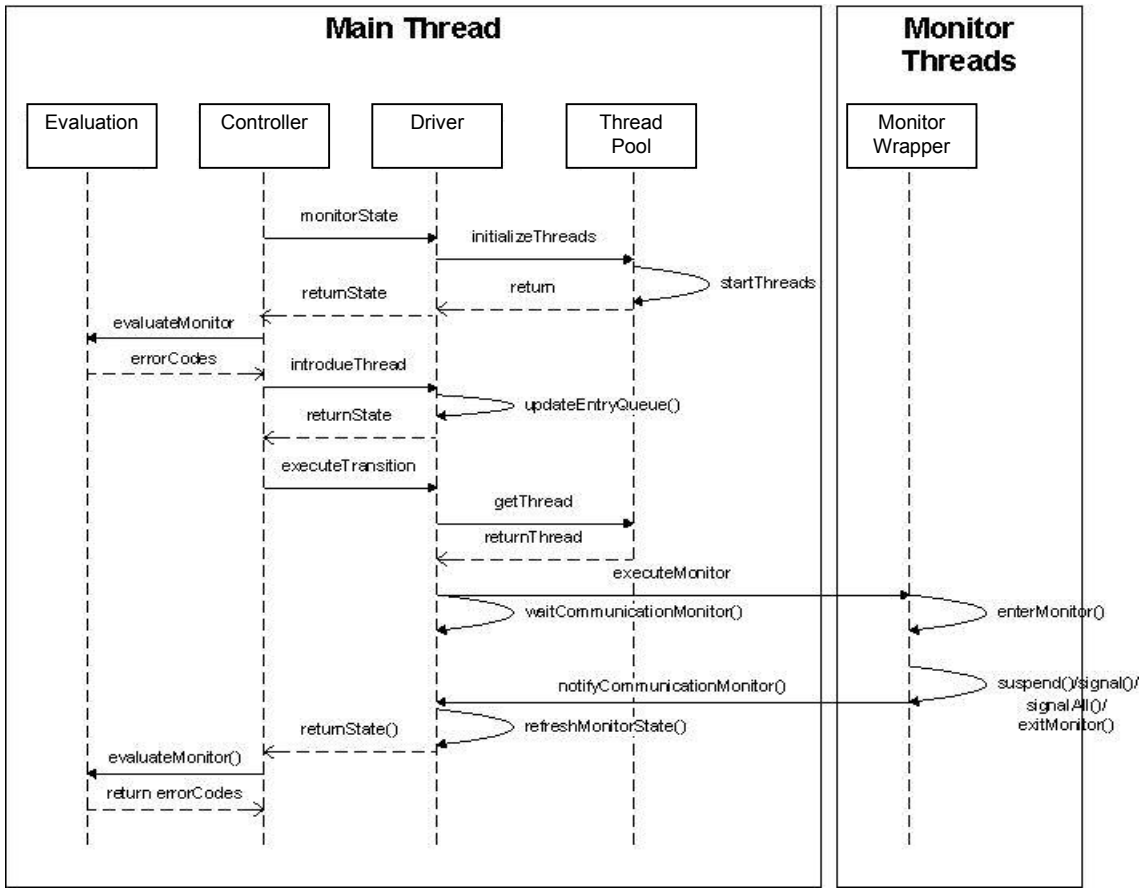

Figure 3.2 Sequence Diagram

There are two major threads of execution: -

- Main Thread: - This is MonitorExplorer thread with four major objects: -
    - Controller: - This is the first object created when execution starts. This creates the Driver and Evaluation object. The controller tells the

Driver what needs to be done and gets the actual monitor state. It extracts abstract state and sends it to the Evaluation.

  - Evaluation: - This get the state from the Controller, evaluates the state and returns if it error or not.
  - Driver: - It gets the transitions to execute from the Controller. It sets up the ThreadPool, gets a thread from it and executes the transition. It returns the actual Monitor state to the Controller. It sets up and interacts with the MonitorWrapper where the actual threads execute the monitor operations.
  - ThreadPool: - This pool of thread creates generic threads has hands one the Driver when required.

- Monitor Threads: - This has the MonitorDriver object which interacts with Driver. There are multiple threads running here which execute the synchronized operations on the Java Monitor.

CHAPTER 4

EXPERIMENTS

In this chapter we look at several experimental results which confirm the effectiveness of MonitorExplorer in detecting synchronization mistakes. We also take a look at the steps required to setup the tool.

4.1 Tool Setup

We need to first establish the requirements that the monitor must fulfill. This is line with traditional testing techniques and must be done independently of the implementation.

In order to use toolbox we need to change the monitor code. The following changes are required for all synchronized methods: -

- Remove the synchronized keyword.
- Add enterMonitor() as the first line.
- Add exitMonitor() as the last line.
- Replace wait() with suspend().
- Replace notify()/notifyAll with suspend()/suspendAll().

We also provide an initialization method for the monitor. This allows the user to test the monitor with different starting conditions. In case of BoundedBuffer Monitor the initialization method allows to specify the size of the buffer and how many slots are filled. Figure 4.1 shows the implementation of eastEnter() method for the FairBridge

Monitor and figure 4.2 show the code for the method after instrumentation so it can be used for the MonitorExplorer tool.

```
1. public synchronized void eastEnter() throws InterruptedException {
2.          ++eastCarsWait;
3.          while (westCars>0 || (westCarsWait > 0 && eastTurn == 0))
4.            wait();
5.          --eastCarsWait;
6.          ++eastCars;
}
```

Figure 4.1 eastEnter() Method

Notice that the synchronized keyword is missing in the instrumented method. Also, enterMonitor() and exitMonitor() methods have been added and wait() has been replaced by suspend().

```
1. public void eastEnter() {
2.          enterMonitor();
3.          ++eastCarsWait;
4.          while (westCars>0 || (westCarsWait > 0 && eastTurn == 0))
5.            suspend();
6.          --eastCarsWait;
7.          ++eastCars;
8.          exitMonitor();
}
```

Figure 4.2 Instrumented eastEnter() Method

We then define the Abstract State. Actually the user only needs to define the abstract values for the data elements. The abstraction for the critical section and condition & entry queue is handled by the tool itself based on the names of the synchronized methods.

We also need to define the evaluation function. We have provided the user API which allows the user to easily define this function. This is used to check if the current abstract state violates any of the requirements. The user can use the current abstract state, the last transition executed and any of the older states or transitions to verify the current state.

The last bit is the input file which has the following format: -

- The first line is the name of the monitor.
- The second line is an integer which can be used to supply the argument for the monitor constructor.
- The third line has the number of synchronized methods followed by two lines for each method. The first of these lines is the name of method and the second is the argument. Incase of no argument we pass the value as -1.
- This block is followed by the number of data members to be tracked. Each data element is represented by two lined. First line has the name and the second has the initial value.

4.2 Mutants

The main objective of conducting these experiments is to establish the effectiveness of our tool. To achieve this goal we introduce some errors, which are commonly found in monitor implementations, and track how many of these errors we can detect. Please note that the objective of our tool is to detect synchronization faults and thus are mutants are focused only common synchronization mistakes. The following guidelines are used to create mutants: -

- Replace while with if statement incase the block contains a suspend() operation.
- Replace signal() with signalAll() and the other way round.
- In case a conditional block contains a synchronization operation change the relational/Boolean operator or use negation of the Boolean variable at the branching point.
- Remove the synchronization operations.

The first mutant detects if the case where we want to recheck the condition when a thread re-enters the critical after it awakened by signal() or signalAll(). The second condition checks if one thread is awakened from the condition queue when we expect all of them to be awakened. The remaining conditions check for conditions where we expect to execute a synchronization operation but that does not happen and also the other way around.

We exercise each mutant independent of other as two mutants may have a masking effect.

4.3 Monitors Tested

We have tested the following monitors: -

- BoundedBuffer: - This monitor solves the Producer/Consumer problem. When the buffer is full, the producer must wait for the consumer to withdraw an item. Similarly, when the buffer is empty the consumer should wait for the producer to produce an item.

- FairBridge: - A solution that prevents collisions in a single-lane bridge. The solution guarantees no starvation. That is, cars from both directions get a fair chance to access the bridge.
- SimpleAllocator: - A solution used to allocate a pool of balls to the golf player. A player must wait if he requests more balls than available.

The BoundedBuffer problem is from [6] and the rest are from [9]. The details of the instrumentation, mutants, requirements, abstraction and results, for each of the monitors, is provided in appendix.

4.4 Experiment Results

Table 4.1 Experiment Results

Monitor	# of Requirements	# of Mutants	# of Mutants Killed	# of Paths Explored	# of Transitions Executed	# of States Explore	Exploration Time (Seconds)
Bounded Buffer	7	12	12	15	47	33	3.2
Fair Bridge	10	22	20	160	386	227	45.4
Simple Allocator	5	14	12	123	248	126	29.7

CHAPTER 5

RELATED WORK

There has been much work done in the area of testing in general but the problems of determining the number of threads and non-deterministic execution does not exist in sequential programs. Our work focuses on testing monitors, thus we review existing work on testing monitors and concurrent programs.

The monitor was proposed by Brinch Hansen and Tony Hoare and Hansen was the first person to propose a solution to test monitors [4]. He suggested an approach to test Pascal monitors with the following steps: -

- The tester identifies a set of preconditions that will cause each branch of operation to be executed at least once.
- Then a single sequence of monitor calls is constructed. This sequence satisfies each identified precondition at least once.
- A test driver is created with multiple threads to execute the monitor call sequence identified in the previous step.

Each of the threads in the test driver executes one or more of the monitor call in the sequence. Also, the threads in the driver are synchronized during testing to ensure they execute in the specified order.

The approach given above was extended to Java Monitors [5]. The work in [5] adds that we identify preconditions that cause each loop to be executed zero, one and

more than one times. This is required as the wait() operation in Java Monitors is often inside a loop.

In [3] [6] Carver and Tai have generalized Hansen's work for synchronizing threads during testing and they have demonstrated how to apply their technique to test monitors, semaphores, locks and message passing. These are the steps suggested by them in [6]: -

- Derive a set of validity constraints from the specification of the program.
- Then perform non-deterministic to determine the coverage and validity.
- Generate additional test sequences for the paths that were not covered and perform deterministic testing for those test sequences.

The works shown above are the only approaches to testing monitors in the literature. Even with tool supports for [3] [5] and [6] the first two steps are manual and error prone. On the other hand, our approach as shown is more systematic and highly automated. Also, we do not rely on loop and branch coverage as these approaches do. We explore the state space of the Java Monitor till we reach a duplicate or error state.

The approaches suggested by Godefroid [7] and Havelund & PressBurger [8] use state space exploration to test concurrent programs. They either directly explore the state space of the concurrent program or extract an abstract model from the program and then explore the abstract model using a model checker.

They problem with these approaches are they do not solve the problem of state explosion. Also, these approaches assume that the program in test is complete. This means that program can be executed in isolation. Thus before we use can test a Java

Monitor a program that uses the monitor under test must be constructed. The problem with this is that the program we construct may not trigger unexpected scenarios which are often the cause of failure. Our approach however is different as we introduce threads on fly, during state space exploration. Also, we impose no restriction on number or type of threads to be introduced. To best of our knowledge, only our work has such capability.

CHAPTER 6

CONCLUSION

This document shows that are approach to testing Java Monitors has successfully used the state exploration technique to test monitors. This undoubtedly will be an asset to developers of Java Monitors in unit testing. The key benefit is the automation of the process thus ensuring greater coverage which is guaranteed neither by current sequential program testing techniques nor by manually creating test sequences and executing them. Our approach also simplifies regression testing of Java Monitors.

The main contribution of our work is introduction of threads on-the-fly. This allows us not to impose restrictions on number or type of threads we introduce. Thus we can generate and execute test sequences on the fly which create race conditions and thus are able to detect more errors. This allows us to detect conditions like starvation, violation of the critical section. An example of this is when the loop condition around the suspend() in FairBridge Monitor is changed to a branch condition, i.e. while is replace by if, we are able to detect violation of the critical section.

The experiment results in chapter 4 show that our tool is able to kill a large percentage of mutants. One mutant that we are not able to kill is when the suspend() is removed from loop condition. However this error will be visible to the user in terms of the system entering a livelock. Another important achievement is that the entire state

space is explored in a small time. Thus the tool can be used for regression without much time penalty.

Our tool also allows us to control the intercept synchronization operation with the help of the toolbox. This allows us to refresh the monitor state on every synchronization operation, thus enabling us to determine the possible transitions at each of these states and explore the state space.

My main contribution to the MonitorExplorer is the MonitorTesting module. It includes the Controller, ThreadPool and Utilities sub-packages. I implemented the MonitorTest algorithm and the getEnabledTransitions() method. I also implemented the logic for retrace, interaction between the user & the system and designed the data structures like Transition and AbstractState. Another important component that I designed and implemented was the communcationMonitor() which enforces synchronization between the main thread and executing threads.

CHAPTER 7

FUTURE WORK

Our work has made significant contributions to testing of Java Monitors. However, we see some enhancements in this future which would make the tool easier to use. The following would be the future enhancements: -

- Automatic instrumentation the Java Monitor code. This would allow the user to provide us with the Java Monitor and the tool would do all the required instrumentation like adding enterMonitor() and exitMonitor() methods.
- A graphical user interface to input the files and setup the tool would be another task in the future. We would also like to present the state space being explored in a graphical form.
- There are some monitors which have a protocol which defines in what order the methods can be executed. An example of this would the Allocator problem. The golfer should only be able to return the balls if he has been given any. The user should be allowed to specify such a protocol.

We would also like to do a comparative study between the performance of the existing approaches and our work.

APPENDIX A

BOUNDED BUFFER EXPERIMENT

Bounded Buffer Experiment

1. Requirements

Requirement #	Requirements 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

Date Members Abstraction

Variable Name: FullSlots

Abstract Value	Actual Value	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, rewithdraw, redeposit, “”}

Critical Section Abstraction: - {withdraw, deposit, “”}

Condition Queue Abstraction: - { {“”, “”}, {withdraw. “”}, {“”, deposit}, …, {withdraw++, “”}, {“”, deposit++}, {withdraw++, deposit++} }

3. Source Code

```
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. Mutant
Replace while with if in withdraw() method

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

5. 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

SIMPLE ALLOCATOR EXPERIMENT

Simple Allocator Experiment

1. Requirements

Requirement #	Requirements Description
1	If one or more than one ball available then pro should not be waiting.
2	If two or more than two balls available then amatuer should not be waiting.
3	If enough balls not available then the request should not be granted.
4	Balls can only be returned by golfer who received them.
5	Golfers should be allowed to return balls.

2. Abstract State

Date Members Abstraction

Variable Name: Available

Abstract Value	Actual Value	Valid
0	available == 0	Y
1	available == 1	Y
2	available == 2	Y
3 – (N -3)	3 <= available <= (N-3)	Y
N -2	available == N - 2	Y
N -1	available == N – 1	Y
N	available == N	Y
N++	available > N	N
0--	available < 0	N

Entry Queue Abstraction: - { get_pro, put_pro, get_am, put_am, reget_pro, reput_pro, reget_am, reput_am, “”}

Critical Section Abstraction: - { get_pro, put_pro, get_am, put_am, “”}

Condition Queue Abstraction: - { {“”, “”, “”, “”}, {get_pro,””,””, “”},{“”, put_pro, ””, “”}, …, { get_pro++, put_pro++, get_am++, put_am++ } }

3. Source Code

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

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

public class SimpleAllocator extends MonitorToolBoxWrapper {

    public int available = 0;

    public int max = 0;

    private int proIn = 0;

    private int amIn = 0;

    public SimpleAllocator(int max, int available, int proIn, int amIn) {
        this.available = available;
        this.max = max;
        this.proIn = proIn;
        this.amIn = amIn;
    }

    public void get_pro() {
        enterMonitor();
        int n = 1;
        while (n > available)
            suspend();
        available -= n;
        proIn++;
        exitMonitor();

    }

    public void put_pro() {
        enterMonitor();
        if (proIn > 0) {
            int n = 1;
            available += n;
        }
```

```
        exitMonitor();
    }

    public void get_am() {
        enterMonitor();
        int n = 2;
        while (n > available)
            suspend();
        available -= n;
        amIn++;
        exitMonitor();

    }

    public void put_am() {
        enterMonitor();
        if (amIn > 0) {
            int n = 2;
            available += n;
            amIn--;
        }
        exitMonitor();
    }
}
```

4. Mutant
Replace signalAll() with signal() in put_pro() method

```
public void put_pro() {
        enterMonitor();
        if (proIn > 0) {
            int n = 1;
            available += n;
            signal(); =➔ signalAll() replaced by signal()
            proIn--;
        }
        exitMonitor();
    }
```

5. Results

Simple Allocator: Please enter the path of input file:
c:\input.txt

…
…
Path: 48 begins....
Path: 49 begins....

---- CONDITION FOUND!!! ----
Condition #: 01 Type: ERROR
Category : starvation
Description: Pro waiting when balls avialable
Requirement: 01 - If one or more than one ball available then pro should not be waiting.
ATTRIBUTES :
Name : available current value: 1
Name : max current value: 10

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 get_pro
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 get_pro
The value of data element is 0
----------Start of Abstract State----------
State #: 4
Thread at the head of the Entry Queue is get_pro
Critical Section is Empty
Thread in the Condition Queue is get_pro
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 get_pro+
The value of data element is 0
----------Start of Abstract State----------
State #: 6
Thread at the head of the Entry Queue is get_am
Critical Section is Empty
Thread in the Condition Queue is get_pro+
The value of data element is 0
----------Start of Abstract State----------
State #: 7
Entry Queue is Empty
Critical Section is Empty
Thread in the Condition Queue is get_pro+
Thread in the Condition Queue is get_am
The value of data element is 0
----------Start of Abstract State----------
State #: 8
Thread at the head of the Entry Queue is get_am
Critical Section is Empty
Thread in the Condition Queue is get_pro+
Thread in the Condition Queue is get_am
The value of data element is 0
----------Start of Abstract State----------
State #: 9
Entry Queue is Empty
Critical Section is Empty
Thread in the Condition Queue is get_pro+
Thread in the Condition Queue is get_am+
The value of data element is 0
----------Start of Abstract State----------
State #: 10
Thread at the head of the Entry Queue is put_pro
Critical Section is Empty
Thread in the Condition Queue is get_pro+
Thread in the Condition Queue is get_am+
The value of data element is 0
----------Start of Abstract State----------
State #: 11
Entry Queue is Empty
Thread in Critical Section is put_pro
Thread in the Condition Queue is get_pro+
Thread in the Condition Queue is get_am+
The value of data element is 1

----------Start of Abstract State----------
State #: 12
Thread at the head of the Entry Queue is get_am
Thread in Critical Section is put_pro
Thread in the Condition Queue is get_pro+
Thread in the Condition Queue is get_am
The value of data element is 1
----------Start of Abstract State----------
State #: 13
Entry Queue is Empty
Critical Section is Empty
Thread in the Condition Queue is get_pro+
Thread in the Condition Queue is get_am+
The value of data element is 1
STATE NOT VALID!!!!!!!!!!!
Path: 50 begins....
Path: 51 begins....

APPENDIX C

FAIR BRIDGE EXPERIMENT

Fair Bridge Experiment

1. Requirements

Requirement #	Requirements Description
1	Cars coming from different directions cannot access the bridge at the same time
2	If no West cars in the bridge or waiting, East cars should be able to access
3	If no East cars in the bridge or waiting, West cars should be able to access
4	East or West cars can only exit once they have entered
5	If at least one East car is in the bridge and no West Car waiting, all east cars should be able to access the bridge
6	If at least one West car is in the bridge and no East Car waiting, all west cars should be able to access the bridge
7	If east car waiting and east car turn then West car should not enter.
8	If west car waiting and west car turn then East car should not enter.
9	Only cars which are waiting or requesting to enter can do so.
10	Cars that have entered should not be in the waiting queue.

2. Abstract State

Date Members Abstraction

Variable Name: westCars and eastCars

Abstract Value	Actual Value	Valid
0;0	westCars == 0 && eastCars == 0	Y
W;0	westCars => 1 && eastCars == 0	Y
0;E	westCars == 0 && eastCars => 1	Y
W;E	westCars => 0 && eastCars => 0	N
-;-	westCars < 0 && eastCars < 0	N
X;-	westCars => 0 && eastCars < 0	N
-;X	westCars < 0 && eastCars => 0	N

Variable Name: westCarsWait and eastCarsWait

Abstract Value	Actual Value	Valid
0;0	westCarsWait == 0 && eastCarsWait == 0	Y
WW;0	westCarsWait > 0 && eastCarsWait == 0	Y
0;WE	westCarsWait == 0 && eastCarsWait > 0	Y
WW;WE	westCarsWait > 0 && eastCarsWait > 0	Y
-;-	westCarsWait < 0 && eastCarsWait < 0	N
X;-	westCarsWait => 0 && eastCarsWait < 0	N
-;X	westCarsWait < 0 && eastCarsWait => 0	N

Variable Name: eastTurn

Abstract Value	Actual Value	Valid
True	1	Y
False	0	Y

Entry Queue Abstraction: - {enterEast, enterWest, exitEast, exitWest, reenterEast, reenterWest, reexitEast, reexitWest, “”}

Critical Section Abstraction: - {enterEast, enterWest, exitEast, exitWest,, “”}

Condition Queue Abstraction: - {{“”, “”, “”, “”}, { enterEast,””,””, “”}, {“”, enterWest,””, “”}, …, { enterEast++, enterWest++, exitEast++, exitWest++} }

3. Source Code

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

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

public class FairBridge extends MonitorToolBoxWrapper{

    public int westCars = 0;
    public int eastCars = 0;
    public int westCarsWait = 0;
```

```
public int eastCarsWait = 0;
public int eastTurn = 1;

public FairBridge(int eastCars, int westCars){
    this.eastCars = eastCars;
    this.westCars = westCars;
}

public void eastEnter() throws InterruptedException {
    enterMonitor();
    ++eastCarsWait;
    while (westCars>0 || (westCarsWait > 0 && eastTurn == 0))
        suspend();
    --eastCarsWait;
    ++eastCars;
    exitMonitor();
}

public void eastExit(){
    enterMonitor();
    if (this.eastCars >0){
        --eastCars;
        if (eastCars==0){
            signalAll();
        }
        eastTurn = 0;
    }
    exitMonitor();
}

public void westEnter(){
    enterMonitor();
    ++westCarsWait;
    while (eastCars>0 || (eastCarsWait > 0 && eastTurn == 1))
        suspend();
    --westCarsWait;
    ++westCars;
    exitMonitor();
}

public void westExit(){
    enterMonitor();
    if (this.westCars >0){
        --westCars;
```

```
            if (westCars==0){
                signalAll();
            }
            eastTurn = 1;
        }
        exitMonitor();
    }
}
```

4. Mutant
Change the condition in the while statement in the enterEast() method

```
      public void eastEnter() {
        enterMonitor();
        ++eastCarsWait;
        while (westCars<0 || (westCarsWait > 0 && eastTurn == 0))
            =➔ Changed westCars > 0 to westCars < 0
            suspend();
        --eastCarsWait;
        ++eastCars;
        exitMonitor();
    }
```

5. Results

…
…
Path: 9 begins....
Path: 10 begins....

---- CONDITION FOUND!!! ----
Condition #: 01 Type: ERROR
Category : collision
Description: West and East Cars are in the bridge
Requirement: 01 - Cars coming from different directions cannot access the bridge at the same time
ATTRIBUTES :
Name : westCars current value: 2
Name : eastCars current value: 1
Name : westCarsWait current value: 0
Name : eastCarsWait current value: 1
Name : eastTurn 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;0
The value of data element is 0;0
The value of data element is true
----------Start of Abstract State----------
State #: 2
Thread at the head of the Entry Queue is eastEnter
Critical Section is Empty
Condition Queue is Empty
The value of data element is 0;0
The value of data element is 0;0
The value of data element is true
----------Start of Abstract State----------
State #: 3
Entry Queue is Empty
Critical Section is Empty
Condition Queue is Empty
The value of data element is 0;E
The value of data element is 0;0
The value of data element is true
----------Start of Abstract State----------
State #: 4
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
The value of data element is 0;0
The value of data element is true
----------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
The value of data element is WW;0
The value of data element is true
----------Start of Abstract State----------

State #: 6
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
The value of data element is WW;0
The value of data element is true
----------Start of Abstract State----------
State #: 7
Entry Queue is Empty
Critical Section is Empty
Thread in the Condition Queue is westEnter+
The value of data element is 0;E
The value of data element is WW;0
The value of data element is true
----------Start of Abstract State----------
State #: 8
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
The value of data element is WW;0
The value of data element is true
----------Start of Abstract State----------
State #: 9
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
The value of data element is WW;0
The value of data element is true
----------Start of Abstract State----------
State #: 10
Thread at the head of the Entry Queue is eastEnter
Thread in Critical Section is eastExit
Condition Queue is Empty
The value of data element is 0;0
The value of data element is WW;0
The value of data element is false
----------Start of Abstract State----------
State #: 11
Thread at the head of the Entry Queue is RewestEnter
Thread in Critical Section is westEnter
Thread in the Condition Queue is eastEnter

The value of data element is W;0
The value of data element is WW;WE
The value of data element is false
----------Start of Abstract State----------
State #: 12
Entry Queue is Empty
Critical Section is Empty
Thread in the Condition Queue is eastEnter
The value of data element is W;0
The value of data element is 0;WE
The value of data element is false
----------Start of Abstract State----------
State #: 13
Thread at the head of the Entry Queue is eastEnter
Critical Section is Empty
Thread in the Condition Queue is eastEnter
The value of data element is W;0
The value of data element is 0;WE
The value of data element is false
----------Start of Abstract State----------
State #: 14
Entry Queue is Empty
Critical Section is Empty
Thread in the Condition Queue is eastEnter
The value of data element is W;E
The value of data element is 0;WE
The value of data element is false
STATE NOT VALID!!!!!!!!!!
Path: 11 begins....
Path: 12 begins....

REFERENCES

[1] http://today.java.net/pub/a/today/2004/08/02/sync1.html

[2] Lei, Jeff (2006). Class notes for Advanced Topics in Software Engineering. http://crystal.uta.edu/~ylei/cse6324/

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

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

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

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

[7] Godefroid, P. "Model Checking for Programming Languages using VeriSoft," Proc. of the 24th ACM Symposium on

[8] Havelund, K. & Pressburger, Tom. "Model Checking Java Programs Using Java PathFinder," International Journal on

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

BIOGRAPHICAL INFORMATION

Vidur Gupta received his M.S. in Computer Science degree from University of Texas at Arlington in May 2006. He received Post Graduate Diploma in Computer Applications from the Institute of Management Technology, India in May 2003 and Bachelors of Commerce with Honors from University of Delhi, India in May 2000. His research interests include testing of concurrent programs, networking and use of computer science in enhancing quality of life.