INTERLEAVING COVERAGE CRITERIA ORIENTED TESTING OF MULTITHREADED APPLICATIONS

by

Mehmet Çağrı Çalpur

Submitted to the Graduate School of Engineering and Natural Sciences

in partial fulfillment of

the requirements for the degree of

Master of Science

Sabancı University

February, 2012

INTERLEAVING COVERAGE CRITERIA ORIENTED TESTING OF MULTITHREADED APPLICATIONS

APPROVED BY:

Asst. Prof. Dr. Cemal Yılmaz	[illegible]
(Thesis Advisor)	
Assoc. Prof. Dr. Albert Levi	[illegible]
Assoc. Prof. Dr. Erkay Savaş	[illegible]
Asst. Prof. Dr. Esra Erdem	[illegible]
Asst. Prof. Dr. Kerem Bülbül	[illegible]

DATE OF APPROVAL: 03/02/2012

INTERLEAVING COVERAGE CRITERIA ORIENTED TESTING OF MULTITHREADED APPLICATIONS

Mehmet Çağrı Çalpur

CS, Master's Thesis, 2012

Thesis Supervisor: Cemal Yılmaz

Keywords: Software Testing, Covering Arrays, Concurrent Programs, Instrumentation, Interleaving Coverage

Abstract

Concurrent programs run several to thousands of processes or threads in parallel and the correctness of the outcome is critical. Successful tests for deterministic systems can not be applied to concurrent programs, because of their non-deterministic behavior. Exhaustive testing is not applicable because of the search space and testing costs. We have designed a testing algorithm that produces Sequence Covering Arrays of a concurrent program's execution segments, and tests these interleaving sequences. We provide a coverage metric that works as a measure to define the ratio of covered test possibilities. Our approach relies on the sequence covering arrays to cover all interleavings, while requiring least amount of testing. This thesis presents the Interleaving Coverage Criteria-oriented testing of multithreaded programs, it's utility programs to take over the control of applications to run tests and the case studies that we have done to show the efficiency of the system against exhaustive testing and its variants.

ÇOK KANALLI UYGULAMALARIN SERPİŞTİRME KAPSAMA KRİTERİYLE TEST EDİLMESİ

Mehmet Çağrı Çalpur

CS, Yüksek Lisans Tezi, 2012

Tez Danışmanı: Cemal Yılmaz

Anahtar Kelimeler: Yazılım Testi, Kapsama Dizileri, Koşut Zamanlı Programlar, Enstrümantasyon, Serpiştirme Kapsama

Özet

Koşut zamanlı programlar binlerce paralel çalışan programdan oluşabilir ve bunların doğru çalışabilmesi çok önemlidir. Başarılı deterministik testler koşut zamanlı programlarda, deterministik olmayan davranışları nedeniyle kullanılamaz. Etraflı testler ise keşfedilemeyecek kadar büyük test uzayına sahip oldukları için pratikte kullanılamamaktadır. Tasarladığımız test algoritması program bölümleri ile eşleşen Düzen Kapsama Dizileri üreterek, serpiştirme düzenlerini test eder. Kapsadığımız test olasılıklarını ölçümleyecek bir test birimi oluşturduk. Bizim yaklaşımımız düzen kapsama dizileri kullanarak az test ile bütün serpiştirmeleri kapsamaktır. Bu tezde çok kanallı uygulamaların serpiştirme kapsama kriteriyle test edilmesi, bu testin yardımcı programları, vaka araştırmaları ve etraflı testler ve türevlerine üstünlüğü anlatılmaktadır.

Acknowledgements

I wish to express my gratitude to my advisor Cemal Yılmaz for proposing this challenging and interesting project and for all of his advice, encouragement and guidance along the way.

Thanks to my family and beloved friends for their love and support.

Contents

List of Figures

List of Tables

1 Introduction

Software testing is one of the most crucial parts of software development life cycle. A software system, especially today's large scale, heavy traffic, mission critical systems, is naturally error-prone. Whether the error is a result of its programmers' mistake, compiler-generated faulty code or a design error, even the smallest error may cause a ripple effect and produce inconvenient results for the users. Unnoticed software errors may have financial or even lethal consequences.

Uncovering software errors (bugs) require extensive testing of the system under certain test inputs repeatedly. The idea of testing with various inputs is to reveal different execution paths, where the program might fail. Reducing non-determinism of a program is required to cover all possible paths. Administering certain test inputs is the key to eliminate the non-determinism in sequential programs.

Software systems usually handle simultaneous stimuli, which requires concurrent processes. However, testing of concurrent programs are not as straightforward as sequential programs. Adapting sequential testing procedures to concurrent programs fails to provide full coverage of the execution paths. The behavior of the program depends on external factors as well as program inputs. And while the scale of software systems exhibit rapid increase, exhaustive testing of every possibility is getting more and more costly, if not impossible.

Concurrent programs utilize parallel working components, these are called processes and threads. Our study focuses on threads, specifically Java threads. Threads in a program use the same memory space for execution, whereas processes run in separate memory spaces. The shared memory model of threads is a useful simplification for the programming practice. On the other hand, parallelism of the program execution results in data integrity issues, race conditions and execution ordering related bugs.

1.1 Motivation

In this thesis we propose a system for testing of multithreaded java programs. We assume that a computer program, that consists of threads that run in parallel, is a collection of execution segments (Blocks). **Critical sections** are the segments, where concurrency affects the behaviour of the system and **non-critical segments** are mutually exclusive. Our system parses these segments, defines them with unique identification information and enumerates the segments in order to build up an execution order to simulate various behaviours of the system with a constant test input.

A concurrent program, even the least complex one, with a small number of blocks require immense number of tests to cover all execution order possibilities to perform exhaustive testing. We believe that applying the sequence covering arrays (SCA) concept into testing computer software would dramatically reduce the number of tests and time required to test a program. By grouping up blocks of a program in sequence covering arrays and concatenating these arrays to build an execution order, our approach achieves very high test coverage ratios.

The first part of the system is a controller/interface, which consists of two sub-components. The first component is a Java bytecode analyzer and instrumenter, which is used to find and modify all concurrency related code of an application. The second component is our concurrency library and scheduling interface. The concurrency library is a modification of Java Platform's Object and Concurrency libraries. Our concurrency methods replace the original methods by utilizing the instrumenter component. The scheduling interface identifies the blocks that are executed and informs our scheduler.

The second part of the system is a scheduler, which processes the block information sent by the Thread Scheduler Interface. The block information is used to keep track of the states of thread interleavings. The scheduler is capable of analyzing the currently available thread state information, executed block information, possible upcoming thread interleaving information and previous block schedules, in order to decide the course of execution. The scheduler's purpose is to dictate the execution order of threads according to some coverage criteria. The scheduler tries to accomplish the coverage criteria and test as many interleavings as possible to observe the concurrent behaviour and expose bugs.

2 Background

2.1 Defects in Multithreaded Applications

2.1.1 Data Race

Data race bugs occur when two threads try to access a shared variable simultaneously without proper synchronization, which means that the shared variable lacks a mutual exclusion mechanism such as a common lock.

2.1.2 Deadlock

A deadlock occurs when a thread enters a waiting state because of a resource requested by the thread is being held by another waiting thread, which also waits for another resource. If the thread is unable to change its state indefinitely because the resources requested by it are being used by other waiting threads respectively, in a circular fashion. When none of the threads have the opportunity to release the locks they previously acquired, then the system is said to be in a deadlock (Figure 1).

2.1.3 Atomicity Violation

A computer code, instruction or a set of instructions, is atomic, when the code can not be interrupted during its execution. Atomicity is achieved in hardware and can be simulated in software. Atomic instructions supported by the hardware is used to implement atomic methods in software. Atomicity violation bugs are caused by concurrent execution unexpectedly violating the

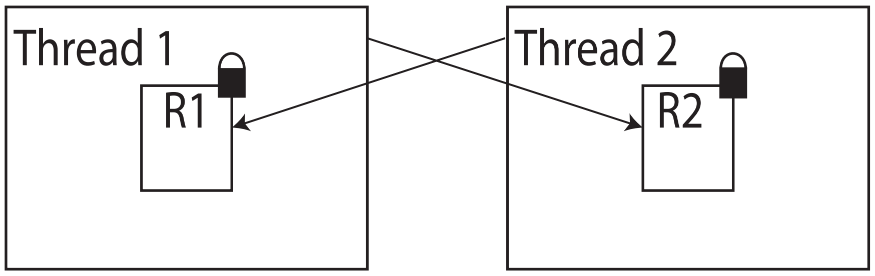
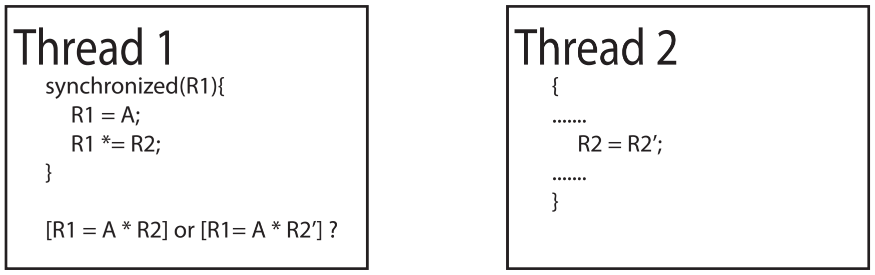


Figure 1: A **deadlock** example where Thread 1 holds the lock for resource 1 and Thread 2 holds the lock for resource 2, both threads are in a waiting state to acquire the lock for the other resource.

atomicity of a certain code region (Figure 2). Atomicity violation bugs do not cause deadlock, but they compromise the integrity of the result of execution.



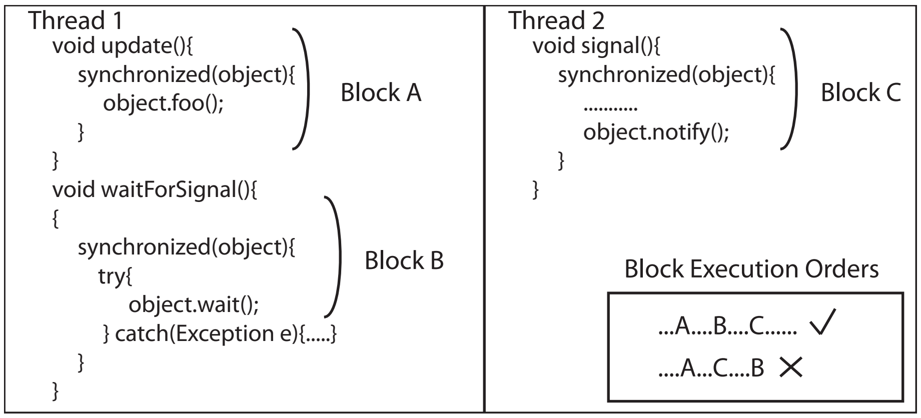
```
synchronized(R1){
    R1 = A;
    R1 *= R2;
}

[R1 = A * R2] or [R1= A * R2′] ?
```

```
{
.......
    R2 = R2′;
.......
}
```

Figure 2: An **atomicity violation** example where programmer assumes synchronizing R1 in Thread 1 will secure the atomicity of the operation. Eventhough the lock is held, a concurrently running thread, Thread 2, may change the contents of the shared resource R2.

2.1.4 Order Violation

A group of program segments can be programmed with the intention to be executed in a specific order. If the desired order between execution blocks can not be enforced, then the result of the execution is bugged. Like the atomicity violation bugs, order violation bugs are associated with the integrity of execution, but they may result in a deadlock situation in case of poorly timed wait() and notify() operations. Figure 3 shows an example of the order violation bug, the situation in the figure shows the ”losing a notify” bug pattern.



```
Thread 1
  void update(){
    synchronized(object){
      object.foo();
    }
  }
  void waitForSignal(){
  {
    synchronized(object){
      try{
          object.wait();
      } catch(Exception e){.....}
    }
  }
```

```
Thread 2
  void signal(){
    synchronized(object){
      ..........
      object.notify();
    }
  }
```

Figure 3: An **order violation** example, bug depends on the scheduling of the threads invoking the methods given in figure. Block Execution Orders box gives two schedules that executes properly and yields to a bug.

2.2 Java Platform

The Java Platform is a popular programming platform, widely used by programmers for commercial and academic purposes. One of the main reasons behind its popularity is the promise of cross-platform usability. The word ”Platform” is used instead of ”Language”, because of the fact that Java offers a programming language, a virtual machine environment to run the programs written in the language, specifications for the implemented concepts and support for various environments such as embedded systems, mobile devices and peripheral devices.

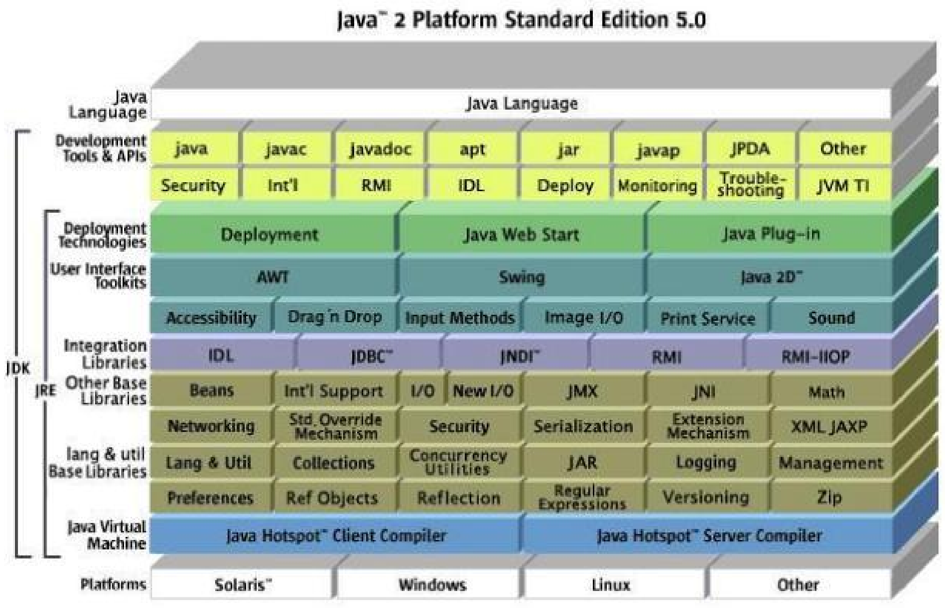


Figure 4: Visual representation of the components constituting the Java Platform

2.2.1 Java Virtual Machine

A virtual machine(VM) is basically an imitation of a cpu or a computer system as a whole, which processes the commands generated by a compiler or interpreter for java programming language. The VM is an abstract system and the programs that run on the VM do not interact with the hardware system directly. The VM, however, must be compatible with the system it is running on. Therefore, there has to be specific VMs for any combination of hardware and operating systems. These VMs all offer the same functionality for the programs running on them. That is the reason for the cross-platform usability.

The JVM is defined by the Java Virtual Machine Specification. The JVM specification defines the bytecode instructions, class file format and the verification algorithm. Java Bytecodes are a set of instructions run by the VM. The Class file is a binary format that constitutes the java bytecode and class structure information. The verification algorithm is used to inspect the programs that will run on the machine for correctness and malicious intent. Programs which fail the verification test are prevented from running, thus protecting the integrity of the VM and the system it runs on.

2.3 Multithreaded Java

Java threads are independent flow of controls that share the same heap memory (Figure 5). In a computer system with multiple CPUs each thread may run simultaneously in its own CPU. In a single CPU environment there are some scheduling algorithms, which controls the execution of threads. Every thread has a priority value and the execution frequency of threads depends on the priorities associated with them.

Java thread model is controlled by the JVM and the `java.lang.Thread` class. `java.lang.Thread` implements the functions that control the cre-

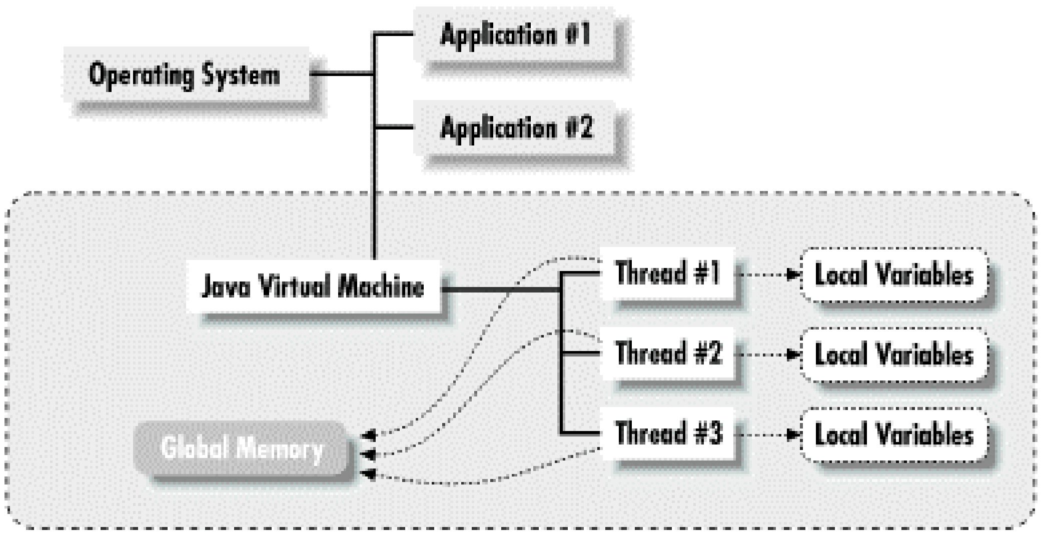


Figure 5: Multithreading Concept of Java Platform

ation, execution and finalization of threads. JVM controls the synchronization of threads by granting access to shared resources and scheduling threads to run.

2.3.1 Thread Scheduling

Thread scheduling is basically organizing the threads' competition for using the CPU. This competition can be regulated by the programmer, the JVM or the operating system. There are various implementations about handling the threads. **Green Threads** model is the simplest and widely used thread implementation of early days of Java. The VM is responsible for the threads and operating system does not interfere. Lately, operating system level thread implementations are more common. Windows Native Threads, Solaris Native Threads and Native Posix Thread Library are examples of operating system level implementations. The variety in this area is caused by the lack of a precisely defined scheduling model by the Java Specification.

2.3.2 Synchronization and Thread Notification

The purpose of synchronization is to coordinate access to shared resources. Multiple threads trying to access a shared resource creates a problem called **race condition**. The `synchronized` keyword acts like a mutex lock and allows the programmer access to a resource. The lock prevents other threads from using the resource as long as the lock is held by a thread. A method, block or a class can be declared `synchronized`. Synchronizing a method is practically the same as synchronizing a block. It is a better practice to keep the synchronization scope as small as possible, in order to prevent synchronization problems.

Each java object has an associated monitor that is used to implement the locking mechanism. The JVM provides two instructions for monitors, `monitorenter` and `monitorexit` are mnemonics for bytecodes that control the locking of an object. When a `synchronized` keyword is used the associated block is bounded by `monitorenter` and `monitorexit` instructions. Only one thread may obtain the monitor of an object and additional locks can be obtained by that thread. Each time a locking thread enters `synchronized` block for an object, the `monitorenter` instruction increases the lock count of the object and each `monitorexit` decreases the lock count by one. The lock can be obtained by a thread when the lock count is 0.

The `java.lang.Object` Class implements the **wait()** and **notify()** methods. These methods are used to control the thread execution that depends on a certain event to occur. In other words, these methods handle the communication between threads. However, these methods can not replace the synchronization mechanism. Thus these concepts should be used in collaboration.

The `wait()` method waits for a condition to occur, and must be called from within a synchronized method or block. Calling the `wait()` method releases the lock of the caller object prior to waiting and reacquires the lock before continuing execution.

The `notify()` method informs a thread that the condition thread is waiting for has occured, and must be called from within a synchronized method or block. When the `notify()` method is called from the object, there is no way to know which thread is notified. The waiting thread that received the notification wakes up and tries to grab the lock and resume execution.

The `notifyAll()` method notifies all the threads waiting on the object that the condition has occured. The method must be called from within a synchronized method or block. The uncertainty of the `notify()` method increases the probability of an erroneous behaviour. The notified thread may wait for another event to occur and execution halts.

2.4 Java Bytecode Instrumentation

Bytecode instrumentation (BCI) is a technique in which bytecode is injected directly into a Java class to achieve some purpose that the class did not originally support. This process has a variety of uses for programmers who want to modify a class without changing the source, or want to change the class definition dynamically at run time for purposes like hotfixing.

2.4.1 A Sample Java Program

```
public class HelloWorld
{
  public static void printMessage()
  {
    System.out.println("Hello World!");
  }
  public static void main(String args[])
  {
    printMessage();
  }
}
(a) HelloWorld.java
```

```
000000 cafebabe           magic = ca fe ba be
000004 0000               minor version = 0
000006 0032               major version = 50

(b) HelloWorld.class File Header
```

```
000008 0025               37 constants
00000a 0a00070016            1. Methodref class #7 name-and-type
    #22
```

```
00000f 0900170018             2. Fieldref class #23 name-and-type
    #24

000024 07001e                 7. Class name #30
000027 010006                 8. UTF length=6
00002a 3c696e69743e                          <init>
000030 010003                 9. UTF length=3
000033 282956                                ()V

0000e5 0c00080009             22. NameAndType name #8 descriptor
    #9
0000ea 07001f                 23. Class name #31
0000ed 0c00200021             24. NameAndType name #32 descriptor
    #33

00011b 010010                 30. UTF length=16
00011e 6a6176612f6c616e672f4f626a656374   java/lang/Object
00012e 010010                 31. UTF length=16
000131 6a6176612f6c616e672f53797374656d   java/lang/System
000141 010003                 32. UTF length=3
000144 6f7574                                out
000147 010015                 33. UTF length=21
00014a 4c6a6176612f696f2f5072696e745374   Ljava/io/PrintSt
00015a 7265616d3b                            ream;

(c) HelloWorld.class Constant Pool
```

```
                              Method 1:
0001e0 0009                       access flags = 9
0001e2 000f                       name = #15<printMessage>
0001e4 0009                       descriptor = #9<()V>
0001e6 0001                       1 field/method attributes:
                                  field/method attribute 0
0001e8 000a                          name = #10<Code>
0001ea 00000025                      length = 37
0001ee 0002                          max stack: 2
0001f0 0000                          max locals: 0
0001f2 00000009                      code length: 9
0001f6 b20002                        0 getstatic #2
0001f9 1203                          3 ldc #3
0001fb b60004                        5 invokevirtual #4
0001fe b1                            8 return
0001ff 0000                          0 exception table entries:
000201 0001                          1 code attributes:
                                     code attribute 0:
```

```
000203  000b                            name = #11<LineNumberTable>
000205  0000000a                        length = 10
                                        Line number table:
000209  0002                            length = 2
00020b  00000004                           start pc: 0 line number: 4
00020f  00080005                           start pc: 8 line number: 5

(d) HelloWorld.class printMessage() Method
```

Figure 6: A sample java program (a) and some parts of its binary file (b), (c), (d)

The bytecode decomposition of a simple java program in Figure 6 shows the main parts of a java `class` file. The first columns in Figure 6 (b), (c), (d) are the hexadecimal indexes of bytes in the binary file. The second column corresponds to the actual bytes in the file in hexadecimal, every two digit is one byte of information. The last column is the human readable interpretation of the `class` file. The data in Figure 6 is generated by the utility program DumpClass.java, that comes with the "Programming for the Java Virtual Machine" book by Joshua Engel.

2.4.2 BCI Libraries

There are various bytecode instrumentation libraries for use, some of them are JBOSS Javassist, Jakarta Bytecode Engineering Library (BCEL) and ObjectWeb ASM. These libraries offer different concepts in their functionality and implementation. In our system JBOSS Javassist was used for handling bytecode instrumentation.

Main use for a bytecode instrumentation library is decoding the binary class file of a java program, gather information about the components of the program and store the information in abstract objects that are more understandable by a user that has little or no knowledge of bytecode instrumentation. They provide simple source code level, java code injection methods. These methods enable the user to add java statement(s) at the start or end of the program. These methods are used to inject **Aspect Oriented Programming** related code into the binary file or generating a new class file from stratch with its variables, contructor and methods.

2.5 JBOSS Javassist Bytecode Instrumentation Library

Javassist is a Java bytecode instrumentation library supported by the JBOSS community. It has been chosen for the project because of the features offered by the library and the ease of use. Our instrumentation program required to work on both source code level and bytecode level. Despite having a few unsupported requirements Javassist provided us to perform the tasks required for the project. The instrumentation program both uses the Javassist library and extends it for some of the missing functionality.

Figure 7 shows a simple example of a target synchronization block (Listing 1) and the source code representation of the program snippet after utilizing our instrumentation program to remove `monitorenter`, `monitorexit` bytecode instructions and replace them with `myMonitorEnter()` and `myMonitorExit()` methods of our scheduler. Javassist's support for byte-

```java
synchronized(object){
  object.foo();
}
```

```java
{
  ThreadScheduler.myMonitorEnter(object, location);

  object.foo();

  ThreadScheduler.myMonitorExit(object , location);
}
```

Figure 7: ByteCode Instrumentation Example

code level manipulation was crucial for tracking down the bytecodes that will be removed from the class file, inserting the new method definitions and inserting the bytecode instructions to invoke these newly added methods from the related method in the original program. The Javassist verification utility was used to determine any errors that might prevent running the instrumented program in the virtual machine.

2.6 Sequence Covering Arrays

Testing a software is generally the most costly part of the software lifecycle, requiring both time and money. In order to accept a system fault-proof, the system must be tested exhaustively by trying every possible combination of options and input. The need to increase the testing performance opened way to a concept called `Covering Arrays`.

Covering arrays are used to break up the testing process of N parameters with m variations into a subset of t elements. These t-wise sequences can be compressed into arrays of N elements. The number of tests required to cover all t-way pairs is significantly lesser than the number of exhaustive tests. A comparison of exhaustive versus sequence covering array testing is given in Figure 8.

$N = 5$
$m = 3$
$t = 3$
# of Exhaustive tests: $m^N = 243$
# of t-way sequences: 2730
# of Sequence Covering Arrays: 18

Figure 8: Sequence Covering Arrays vs Exhaustive Testing

3 Related Works

3.1 Exhaustive Testing

Exhaustive testing of software for all possible inputs and is a conclusive way of testing the correctness. However it has very little practical use, due to the infeasibility of testing vast amount of input space. Coppit et. al. proposed bounded exhaustive testing [14]. In bounded exhaustive testing, the system is tested for all inputs up to some level. But they argue that the system they tested still couldn't handle large data sets. Kuhn et. al. proposes the Pseudo-Exhaustive Testing, which is based on empirical observation of fault triggering variables [15]. They introduce implementing Covering Array concept to test for all conditions generated by the subsequnces of variables.

3.2 Reachability Testing

The non-deterministic behaviour of concurrent programs and the cost of exhaustive testing forced researchers to restrict the number of tests to expose software errors. Hwang et. al. [11] presented a combination of non-deterministic testing and deterministic testing, which is called **reachability testing**. If a program with a specific input contains a finite number of execution blocks, performing the test with same setup many times would lead to an exhaustive testing of the program that can reach all possible states.

3.3 Concurrency Testing with BCI

Java Bytecode Instrumentation is a popular approach for researchers in Software Testing and Aspect-Oriented Programming areas [6, 7, 8, 26]. BCI proves itself to be a powerful tool for providing flexibility of using real Java applications for testing without the need for the source code. Baur proposed recording of program executions that lead to a software failure and with the help of bytecode instrumentation replays the execution to reproduce the software error [6]. Bruening proposed the ExitBlock algorithm, which is a basis for our Thread Scheduler's atomic execution blocks. ExitBlock algorithm analysis a concurrent program to observe all possible behaviors [7]. Bounds implements an instrumentation system to test an application's performance [8]. Gschwind et. al. [26] uses bytecode instrumentation for gathering detailed information of a program's execution trace and object manipulation.

4 The Thread Scheduler

Thread Scheduler Interface is a software package that needs to be installed in the machine where the program that is going to be tested is running. It consists of a Thread Scheduler program and instrumentation program written in Java, javassist BCI library .jar file and a test initiator script that registers our test environment to the Interleaving Coverage Criteria-oriented Tester system and runs the instrumented program to be tested for software errors.

The **Thread Scheduler** program oversees the execution of the multithreaded application with the information it receives by the injected synchronization, concurrency and thread related methods implemented in Thread Scheduler. The Thread Scheduler dictates its own scheduling schema to the running multithreaded application. It receives this scheduling schema from the Testing System and forces the specified thread to run and halt according to the schedule.

4.1 Mutual Exclusion Principle

Mutual exclusion principle is associating a shared object with a lock object and letting at most one thread to obtain the object's lock(s) during the execution of a program. Our Thread Scheduler Interface assures this mutual exclusiveness by injecting its own synchronization and communication methods into the program. Once the target program is instrumented, the thread scheduler keeps track of the locks of an object and updates the locks' status every time `myMonitorEnter()` and `myMonitorExit()` methods are called. The Thread Scheduler Interface also regulates the execution of threads and assures that only one thread is active during an execution of an atomic block. These properties proves that the Thread Scheduler fully undertakes the scheduling and synchronization duties of the JVM.

4.2 Atomic Execution Blocks

In this thesis, we define an **atomic block** as the code segment that starts from a point where thread execution resumes and the point the thread's execution is finished when reaching a `monitorexit` or end of `run()` method. By this definition an atomic block may include both critical and non-critical code segments and the base rule is reaching the method that releases the lock of the synchronized object. The atomicity of these segments are assured by our **Thread Scheduler** implementation. As it was mentioned in the previous section, when a thread is scheduled to run, and released to execute by the Thread Scheduler, it can execute without interruption until reaching either the release of a lock for a synchronized object by invoking `myMonitorExit()` or end of execution. End of execution is tracked by the `threadIsAboutToEnd()` method.

Atomic blocks are also the deciding factor for the coverage criteria of our Testing System. In order to generate the thread interleavings and use these interleavings to cover more unchartered thread schedules, we generate t-way sequence covering arrays of atomic blocks and keep track of the number of covered sequences. The rest of the uncovered sequences are used to generate Thread Schedules that will help increase the coverage. This concept will be explained in Chapter 5.

4.2.1 Atomic Block Decomposition Example

Figure 9 demonstrates an example of parsing atomic blocks in a thread execution lifetime. The start and end points of each atomic block are marked with an informative comment in the figure.

```
public void run() {
  // Start of Block 1
  ThreadScheduler.threadHasStarted(location);
  ThreadScheduler.myMonitorEnter(lock, location);
  lock.update();
  ThreadScheduler.myNotify(lock);
  try {
    ThreadScheduler.myWait(lock, location);
  // End of Block 1
  // Start of Block 2
  } catch (InterruptedException e) {
    e.printStackTrace();
  }
  ThreadScheduler.myMonitorExit(lock, location);
  // End of Block 2
  // Start of Block 3
  ThreadScheduler.threadIsAboutToEnd(location);
  // End of Block 3
}
```

Figure 9: Atomic Blocks of a Thread.

`Atomic block 1` starts from the first line of the `run()` method. The thread is at a waiting state at this point in execution. When the Thread Scheduler gives permission to the thread, it executes until the Thread Scheduler's `myWait()` method is invoked. `myMonitorExit()` is nested in this method, so there is no inconsistency for the end of Atomic Block 1, it ends the execution of an atomic block at the time of releasing the synchronized object's lock.

`Atomic block 2` starts from the end of atomic block 1, where the execution stopped with `myMonitorExit()`. `myMonitorExit()` returns to `myWait()` method and then `myMonitorEnter()` is invoked to regain the object's lock and the blocked thread is once again released to execute. We have defined the boundaries for atomic blocks in the previous section. An atomic block end when the call to release a lock is invoked or the thread execution ends. In Figure 9, even though the critical section ends at the end of atomic block 2, there is still code to be executed. The thread need to be scheduled again in order to resume execution from the end of atomic block 2 to execute the last line of code that finishes the `run()` method. This block becomes the atomic block 3.

The details of the Thread Scheduler's methods and how they control the execution of threads will be thoroughly explained in their respective sections.

4.3 Thread Scheduler Algorithm and Implementation

The **Thread Scheduler** is a library of methods that override synchronization and thread controlling methods in `java.lang.Thread` class and `java.lang.Object` class. The `instrumentFiles.java` program of the Thread Scheduler package is used on the pre-compiled class files of pro-

Class Name	Thread Name	Line # of Code
TwoStage	Thread-0	27

Table 1: Execution information message gathered from the injected code in a tested program. The injected method was executed at line 27 of the specified class.

gram to be tested. This program instruments the bytecode of the program by adding the necessary control methods in the tested program and removing the original methods that is used by the JVM to control synchronization and Thread execution. `instrumentFiles.java` program will be explained in detail in the following sections.

The Thread Scheduler receives execution related data from the tested program each time an atomic block starts and ends the execution. The methods that deliver these data differ by the job they need to perform but the contents of execution data itself is a constant with a strict format. Table 1 shows the structure of this message that contains execution information. The execution information is the concatenation of the `class` file information that the thread recently executed instructions from, the name of the thread executing and the line number of the code that this instrumented method is called.

The Thread Scheduler works as the intermediary system (See Figure 10). It is responsible for delivering the execution information gathered from the test environment to the Tester. The Tester processes this information and applies the coverage criteria to the scheduling process in order to produce a schedule that will achieve better coverage of untested interleaving arrange-

```
BEGIN:
      FOREACH (new Thread i initialized)
            register thread i
            send newThread info to Coverage-oriented Tester
      END FOREACH

      startTesting();

      WHILE (Unfinished threads remaining)
            getNextThreadToSchedule();
            sync(); //Handle Synchronization
            runCurrentThread();
      END WHILE

      endTesting();
END
```

Figure 10: The Thread Scheduler Algorithm. The algorithm shows how the threads are controlled in the testing environment. Thread Scheduler is the intermediary system, delivering messages from the testing environment and executing the orders of the tester.

ments. The overridden synchronization and threading methods are explained in the next section.

4.3.1 Thread Scheduler Methods

BlockThread method (Figure 11) and **UnblockThread** method (Figure 12) are the two methods used to control the execution of the registered threads. A dummy object of the `MyThreadInfo` class is used as the synchronization object. A thread under the control of the Thread Scheduler waits on this object if **BlockThread** method is called and resumes execution if the **UnblockThread** method is called. These methods are not instrumented into the target application, and are used internally in the Thread Scheduler methods.

```java
public static void blockThread(MyThreadInfo thread) {
  synchronized (thread.lock) {
    while (thread.blocked) {
      try {
        thread.lock.wait();
      } catch (InterruptedException e) {
        System.out.println(e.getMessage());
      }
    }
    thread.blocked = true;
  }
}
```

Figure 11: BlockThread Method

```java
public static void unblockThread(MyThreadInfo thread) {
  synchronized (thread.lock) {
    thread.blocked = false;
    thread.lastEntryLocation = thread.nextEntryLocation;
    thread.nextEntryLocation = null;
    currentThread = thread;
    thread.lock.notifyAll();
  }
}
```

Figure 12: UnblockThread Method

The `MyThreadInfo` object is the utility object of the Thread Scheduler. It contains the information used by the rest of the methods of the Thread Scheduler.

Sync method (Figure 13) is the main controller method. The method is responsible for synchronization of threads by calling **BlockThread** on the currentThread and unblockThread for the next thread in the scheduling order. **Sync** method also invokes the **NextThreadToSchedule** method, which communicates with the Tester, asking for the thread to be scheduled next. When testing is finished **sync** method informs the Tester to finalize the test and complete the calculations.

NextThreadToSchedule method (Figure 14) exchanges information with the Tester. The availability information of the registered threads are sent to update the Tester's understanding of the tested application's state. In return, Tester decides which thread will receive permission to run and informs the Thread Scheduler. The method then returns the scheduled thread's information to **sync()** method.

ThreadHasStarted method (Figure 15) is a method injected into the tested application's threads' first line of the **(**run()**)** method. Correctness of the bytecode injection operation for this method is very important, because this method is a thread's first contact point with the Thread Scheduler and the Tester. The thread is registered to the Tester and Thread Scheduler system. The execution of the newly started thread is blocked here at the start of the run() method and stays blocked until the Tester decides to schedule this thread for execution for the first time.

ThreadIsAboutToEnd method (Figure 16) is a method injected into the tested application's threads' last line of the **(**run()**)** method. The invoking thread's execution is about to be finished. This method is responsible for releasing any threads that have previously joined this thread and waiting for it to finish execution. These joined threads are once again free to be

```java
public static void sync(String block) {
  if(testJustStarted){
    testJustStarted = false;
    try {
      // Wait for a short time for the threads
      // to register at the start of a test
      Thread.sleep(2000);
    } catch (InterruptedException e) {
      e.printStackTrace();
    }
  }
  MyThreadInfo nextThread = nextThreadToSchedule(block);

  // if we are finished
  if (nextThread.thread == null) {
    System.exit(0);
  }
  // the same thread is selected
  // so let it run
  if (nextThread.thread == currentThread.thread) {
    currentThread.lastEntryLocation = currentThread.
        nextEntryLocation;
    currentThread.nextEntryLocation = null;
    return;
  }
  // save the current thread
  MyThreadInfo previousThread = currentThread;
  // unblock the next thread
  unblockThread(nextThread);
  // block the currentThread
  blockThread(previousThread);
}
```

Figure 13: Sync Method

```
public static MyThreadInfo nextThreadToSchedule(String block) {

  MyThreadInfo tempThread = NO_THREAD;
  StringBuffer updateBuffer = new StringBuffer();
  for(Iterator it = myThreads.values().iterator(); it.hasNext();
      ){
    tempThread = (MyThreadInfo)it.next();
    int available = 0;
    if(!tempThread.waitingForNotify)
    {
      available = 1;
    }
    updateBuffer.append("#setThreadAv|"+userName+progName+"|"+
        tempThread.thread.getName()+"|"+available);
  }
  if(updateBuffer.length() == 0){
    sendMessage(updateBuffer.toString());
  }else{
    String updateResult = sendReceiveMessage(updateBuffer.
        toString());
  }
  String returned = "NO_THREAD";
  // Get the newly scheduled thread
  returned = sendReceiveMessage("getNext|"+userName+progName+"|"
      +block);

  return (returned.equals("NO_THREAD")) ? NO_THREAD : getTINFO(
      returned);
}
```

Figure 14: NextThreadToSchedule Method

```java
public static void threadHasStarted(String location) {
  // get the current thread which called this method
  Thread newThread = Thread.currentThread();

  // register the thread
  MyThreadInfo newThreadInfo = new MyThreadInfo(newThread);
  newThreadInfo.nextEntryLocation = location;

  // Register the thread to the Tester Program
  sendMessage("regThread|"+userName+progName+"|"+newThread.
      getName()+"|1");
  // Register the thread to the Scheduler
  myThreads.put(newThread, newThreadInfo);

  // block it right away
  blockThread(newThreadInfo);
}
```

Figure 15: ThreadHasStarted Method

	Class Name	Thread Name	Line # of Code
Start	TwoStageThread	Thread-0	-1
End	TwoStage	Thread-0	27

Table 2: Atomic Block information format. This information is parsed to define the atomic block that has been recently executed. It is stored as a thread interleaving of Thread-0. Line number of -1 is the start of run() method.

scheduled by the Thread Scheduler. The thread is unregistered from the Tester and the Thread Scheduler system. This method also marks the end of an atomic block so the atomic block information (See Table 2) is created and the **sync()** method is called to continue testing by scheduling a new thread or finish testing if no more threads are registered.

```
public static void threadIsAboutToEnd(String location) {
  Thread thread = Thread.currentThread();
  MyThreadInfo threadInfo = myThreads.get(thread);

  // Release the threads which have joined current thread
  if(threadInfo.joiners.size()>0){
    while(!threadInfo.joiners.isEmpty()){
      MyThreadInfo joiner = (MyThreadInfo) threadInfo.joiners.
          remove(0);
      if (joiner.waitingForNotify == true) {
        joiner.waitingForNotify = false;
      }
      else{
        System.out.println("Bug! execution should not enter this
             block");
      }
    }
  }
  // Combine the boundary locations of the atomic block to form
      up
  // Atomic Block Information
  String block = threadInfo.id + ": " + threadInfo.
      lastEntryLocation
  + " $ " + location;
  // Unregister the thread from the Tester
  sendMessage("unregThread|"+userName+progName+"|"+thread.
      getName());
  // Unregister the thread from the Thread Scheduler
  myThreads.remove(thread);

  // sync
  sync(block);
}
```

Figure 16: ThreadIsAboutToEnd Method

myMonitorEnter method (Figure 17) is injected into the tested application to replace the original bytecodes generated for synchronized keyword. Javassist library was instrumental for this process. The library enabled us to locate the original bytecode segment used to achieve synchronization. These bytecodes are switched with **nop** instructions (Figures 18, 19), which means "no operation". These instructions are required for the changes to pass the Java verification algorithm. Comparing the indented parts of original and instrumented bytecode segments in (Figures 18, 19) shows that the rest of the code remains untouched to execute properly. After that new function call is injected into the bytecode of the method that is being instrumented. This involves both changing the contents of the method and the constant pool of the class, so that the required classes, variables and definitions are included.

myMonitorEnter method has two types of usage, first type is already mentioned; injection into the tested code. The second type of usage is the Thread Scheduler's own use by invoking the method from other thread control related methods (see Figure 21) to gather the lock for the synchronized object and resume execution of the scheduled thread. For example, **myWait()** method's last method invocation before returning is **myMonitorEnter()** to release the thread had halted in the myWait() method.

```
public static void myMonitorEnter(Object lock, String location)
    {
  // find the lock
  MyLockInfo myLockInfo = myLocks.get(lock);
  if (myLockInfo == null) {
    myLockInfo = new MyLockInfo(lock);
    myLocks.put(lock, myLockInfo);
  }

  // find the thread holding the lock
  Thread threadHoldingIt = myLockInfo.currentlyHeldBy.thread;

  // current thread will block on a lock
  if ((threadHoldingIt != null) && (currentThread.thread !=
      threadHoldingIt)) {
    currentThreadWillBlock();
  }

  // Otherwise go ahead and acquire the lock
  myLockInfo.currentlyHeldBy = currentThread;
  myLockInfo.lockCount++;

  // Current thread will resume its execution
  if (currentThread.interrupted) {
    currentThread.thread.interrupt();
    currentThread.interrupted = false;
    // throw new InterruptedException();
  }
}
```

Figure 17: myMonitorEnter Method

```
                            Method 1:
000472 0001                     access flags = 1
000474 0025                     name = #37<run>
000476 0026                     descriptor = #38<()V>
000478 0001                     1 field/method attributes:
                                field/method attribute 0
00047a 0020                        name = #32<Code>
00047c 000002cb                    length = 715
000480 0003                        max stack: 3
000482 0006                        max locals: 6
000484 000001bd                    code length: 445
000488 b20005                      0 getstatic #5
00048b bb0006                      3 new #6
00048e 59                          6 dup
00048f b70007                      7 invokespecial #7
000492 1208                        10 ldc #8
000494 b60009                      12 invokevirtual #9
********
  0004b3 2a                          43 aload_0
  0004b4 b40003                      44 getfield #3
  0004b7 59                          47 dup
  0004b8 4c                          48 astore_1
  0004b9 c2                          49 monitorenter
0004ba 2a                          50 aload_0
0004bb b40003                      51 getfield #3
0004be b6000f                      54 invokevirtual #15
0004c1 b20005                      57 getstatic #5
0004c4 bb0006                      60 new #6
```

Figure 18: Original bytecode of the run() method of a thread. Indented region is where the object to be synchronized is loaded and lock is acquired by the monitorenter instruction.

```
000d60 12b6                         817 ldc #182
000d62 b600b5                       819 invokevirtual #181
000d65 b800b8                       822 invokestatic #184
000d68 b600a1                       825 invokevirtual #161
000d6b b600b5                       828 invokevirtual #181
000d6e 12b9                         831 ldc #185
000d70 b600b5                       833 invokevirtual #181
000d73 b800b8                       836 invokestatic #184
000d76 b600bb                       839 invokevirtual #187
000d79 04                           842 iconst_1
000d7a 32                           843 aaload
  000d7b b600bd                       844 invokevirtual #189
  000d7e b600bf                       847 invokevirtual #191
  000d81 b600c1                       850 invokevirtual #193
  000d84 b800c4                       853 invokestatic #196
  000d87 00                           856 nop
  000d88 00                           857 nop
  000d89 00                           858 nop
  000d8a 00                           859 nop
  000d8b 00                           860 nop
  000d8c 00                           861 nop
  000d8d 00                           862 nop
000d8e 2a                           863 aload_0
000d8f b40003                       864 getfield #3
000d92 b6000f                       867 invokevirtual #15
000d95 b20005                       870 getstatic #5
000d98 bb0006                       873 new #6
000d9b 59                           876 dup
```

Figure 19: Instrumented bytecode of the run() method of a thread. Indented region is where the myMonitorEnter method is executed. “invokestatic #196” is the instruction representing the method call. The other indented invocations are for generating the parameter passed to the myMonitorEnter method. “nop” instructions are where the original code for synchronization was.

myMonitorExit method (Figure 20) is also injected into the tested application to replace the original bytecode for `monitorexit`. The instrumentation for this method is more complex than instrumenting the code for replacing `monitorenter`. There is one entry point for a synchronized block, while there are various exit points. This is due to the fact that in case of an exception, which may also be coded separately to catch different versions, there has to be a lock release to handle the exception without deadlocking the system. In the instrumentation all `monitorexit` instructions are replaced with **nop** instructions, because the Thread Scheduler is the only synchronization authority and assures atomicity of the blocks and mutually exclusively running of threads. There is no need to gain a native monitor for the synchronized objects, the Thread Scheduler handles its own monitor system. The rest of the exception catching mechanism remains intact and performs accordingly.

myWait method (Figure 21) is the ThreadScheduler's interpretation of the `java.lang.Object` class' wait method. The thread's state is changed to waiting and **myMonitorExit** is called to block the thread's execution. When another thread notifies the lock object and the Thread Scheduler selects the waiting thread to run, **myMonitorEnter** is called and the thread resumes execution. Figures 22, 23 explicitly shows the bytecode changes for the **wait()** and **myWait()** methods. Original **myWait** method has a timed version which has the same functionality, but the thread waits for a specified

```
public static void myMonitorExit(Object lock, String location) {
  // find the lock
  if(lock!= null){
    MyLockInfo myLockInfo = myLocks.get(lock);

    // decrement the lock count
    myLockInfo.lockCount--;
    if (myLockInfo.lockCount == 0) {
      // No body is holding the lock
      myLockInfo.currentlyHeldBy = NO_THREAD;
    }
  }
  currentThread.nextEntryLocation = location;

  // get the block executed

  String block = currentThread.id + ": "
  + currentThread.lastEntryLocation + " $ " + location;

  // Now schedule the next thread
  sync(block);
}
```

Figure 20: myMonitorExit Method. The working of Thread Scheduler's locking mechanism is clearly seen here. The concept is similar to the original methods of the JVM. A lock can be acquired multiple times by its holder and here monitorexit operation makes sure the lock count is decreased properly as intended.

```
public static void myWait(Object lock, String location)
 throws InterruptedException {

  // Change the waiting flag to true
  currentThread.waitingForNotify = true;

  // Add CurrentThread to the list of threads waiting on lock
      object
  MyLockInfo myLockInfo = myLocks.get(lock);
  myLockInfo.addThread(currentThread);

  // Release the lock and sync for another thread
  myMonitorExit(lock, location);
  if (currentThread.interrupted) {
    // currentThread.thread.interrupt();
    // currentThread.interrupted = false;
    throw new InterruptedException();
  }
  // myNotify() or myNotifyAll() method is called and
  // ThreadScheduler rescheduled thread
  // Re-acquire lock and resume execution
  myMonitorEnter(lock, location);
}
```

Figure 21: myWait Method. A thread calls myWait() to release the lock of the synchronized object. This is an exit point for an atomic block. When the waiting thread is rescheduled to run, this point will the starting point of the new atomic block.

amount of time before reentering the competition to regain the synchronized object's monitor.

```
000430 b60009                   45 invokevirtual #9
000433 b6000c                   48 invokevirtual #12
000436 b6000d                   51 invokevirtual #13
000439 2a                       54 aload_0
00043a b40003                   55 getfield #3
00043d b6000e                   58 invokevirtual #14

  // original wait() method call from the object
  000440 2a                       61 aload_0
  000441 b40003                   62 getfield #3
  000444 b6000f                   65 invokevirtual #15
000447 a70008                   68 goto 76
00044a 4e                       71 astore_3
00044b 2d                       72 aload_3
00044c b60011                   73 invokevirtual #17
00044f 840201                   78 iinc 2 1
000452 a7ffba                   79 goto 65545
000455 2b                       82 aload_1
000456 c3                       83 monitorexit
000457 a7000a                   84 goto 94
00045a 3a04                     87 astore 4
00045c 2b                       89 aload_1
00045d c3                       90 monitorexit
00045e 1904                     91 aload 4
000460 bf                       93 athrow
000461 b1                       94 return
000462 0003                     3 exception table entries:
000464 003d                     start pc = 61
000466 0044                     end pc = 68
000468 0047                     handler pc = 71
00046a 0010                     catch type = 16
```

Figure 22: Bytecode decomposition of the run() method of an original example program.

```
000919 04                      230 iconst_1
00091a 32                      231 aaload
00091b b6007b                  232 invokevirtual #123
00091e b6007e                  235 invokevirtual #126
000921 b60080                  238 invokevirtual #128
  000924 b80086                  241 invokestatic #134
  000927 a7000d                  244 goto 257
  00092a 3a05                    247 astore 5
  00092c 1905                    249 aload 5
  00092e b60088                  251 invokevirtual #136
  000931 a70003                  254 goto 257
  000934 00                      257 nop
  000935 00                      258 nop
  000936 00                      259 nop
  000937 00                      260 nop
  000938 00                      261 nop
  000939 00                      262 nop
  00093a 00                      263 nop
00093b a70008                  264 goto 272
00093e 4e                      267 astore_3
00093f 2d                      268 aload_3
000940 b60011                  269 invokevirtual #17
000943 840201                  274 iinc 2 1
000946 a7ff6a                  275 goto 65661
```

Figure 23: Bytecode decomposition of the run() method of the instrumented example program. Indented block is the instrumented call of myWait() and the wait()-nop switch.

myNotify method (Figure 24) is the Thread Scheduler's interpretation of the `java.lang.Object` class' notify method. Original JVM implementation of this method makes a random thread selection from the waiting threads list of the synchronized object. The Thread Scheduler version has the same functionality. This method is responsible for readying a thread for execution by changing its state, but does not initiate execution. In order to execute a notified thread, the Thread Scheduler should chose to schedule this ready-to-run thread. **myNotifyAll** method works the same way as the **myNotify**, but all of the threads on the waiting list of the lock is notified to be ready.

myInterrupt method (Figure 25) is the Thread Scheduler's interpretation of the `java.lang.Thread` class' interrupt method. This method interrupts the thread waiting on any kind of blocking method implemented in `java.lang.Thread` or `java.lang.Object` and throws `InterruptedException`.

myJoin method (Figure 26) is the Thread Scheduler's interpretation of the `java.lang.Thread` class' join method. A running thread joining to another thread blocks itself until the joined thread has finished. A joined thread can be interrupted.

myYield and **mySleep** methods are dummy methods that does not affect the execution under the control of the Thread Scheduler. Implementing the functionality of these methods would contradict with the mutually exclusive execution of threads and the atomic block definitions of the Thread Scheduler algorithm. Yield method simply skips the execution of the cur-

```java
public static void myNotify(Object lock) {
  MyLockInfo myLockInfo = myLocks.get(lock);

  Vector waiting = new Vector();

  MyThreadInfo tempThread = null;
  for (Iterator it = myThreads.values().iterator(); it.hasNext()
      ;) {
    tempThread = (MyThreadInfo) it.next();
    // Check for the waiting state
    // Store these threads for selection in the next part
    if (tempThread.waitingForNotify == true
        && myLockInfo.waitingThreads.contains(tempThread)) {
      waiting.add(tempThread);
    }
  }

  Random r = new Random(System.currentTimeMillis());
  tempThread = (MyThreadInfo) waiting.get(r.nextInt(waiting.size
      ()));
  // Change thread state
  tempThread.waitingForNotify = false;
  // Remove the selected thread from the waiting list
  myLockInfo.removeThread(tempThread);
}
```

Figure 24: myNotify Method

```java
public static void myInterrupt(Thread thread) {
  MyThreadInfo tempThread = myThreads.get(thread);
  MyLockInfo myLockInfo = searchLocks(tempThread);

  if (tempThread.waitingForNotify == true && myLockInfo != null)
      {
    tempThread.waitingForNotify = false;

    myLockInfo.removeThread(tempThread);
  }
  tempThread.interrupted = true;
}
```

Figure 25: myInterrupt Method. An interrupted thread's waiting status is reset to ready and an Interrupted Exception is thrown.

```java
public static void myJoin(Thread thread, String location)
throws InterruptedException
{
  currentThread.waitingForNotify = true;

  MyThreadInfo threadInfo = myThreads.get(thread);
  threadInfo.addThread(currentThread);

  // sync for another thread
  myMonitorExit(null, location);

  if (currentThread.interrupted) {
    throw new InterruptedException();
  }
}
```

Figure 26: myJoin Method. A running thread that calls join changes its status to waiting. This status is reset when the joined thread is finished.

rently running thread without a proper waiting mechanism, the thread is free to run if scheduled immediately after yield. The original implementations of these methods do not release the lock of an object and our implementation requires a myMonitorExit to define an atomic block. The bytecode is instrumented to replace these methods with the dummy methods.

4.4 Instrumenting Files

The Thread Scheduler is a library of methods that enables controlling a computer program written in Java language. In this thesis, our purpose is to control the program in order to test for the concurrency bugs, covering as much of the atomic block interleavings as possible and exposing the errors. The Test System is used on the pre-compiled applications. Therefore we need to modify the application at the bytecode level, so the Thread Scheduler becomes operational and takes over the thread control and synchronization processes. `InstrumentFiles.java` is the utility program that automates the instrumentation process. The program processes all `class` files of the application and instruments them. `InstrumentFiles.java` uses the Javassist Bytecode Instrumentation Library for this purpose, the library is extended for special needs of our testing system.

4.4.1 Interpreting and Running Java Programs

The java source code needs to be compiled into a special kind of file (`class`) that contains special instructions. The JVM understands the file format and interprets the bytecode to the platform depended machine code. A program is executed in the JVM by loading the `class` files of the application into the virtual machine and executing the instructions. The JVM verifies the integrity of these files before executing a program in order to protect the computer system. Javassist BCI library provides methods that check the code and automatically updates the files. Important structural information like program counters, code length information, constant pool entries, exception tables and jump instructions are updated by the library, so that the instrumented application runs in the VM.

4.4.2 Implementation

Javassist implements a ClassPool object that adds all the class files in a specified directory. First of all `InstrumentFiles` program creates a ClassPool object and adds all the instrumentation target `class` files in the pool. The Thread Scheduler is a set of methods that is invoked from inside the tested programs. A java program must import any external library in order to use the methods defined in them. The first task of instrumentation is importing the Thread Scheduler package into the ClassPool object, by calling `importPackage(String packageName)` method. This method forces all files to import the specified package. The `ProcessClassFile()` method is the top level instrumentation method. This method is called for every `class` file in the pool.

ProcessClassFile method (Listing 27) starts the process by extracting the class information into a CtClass object. Then this class' type is checked out to filter out the abstract and interface classes, which are not instrumented by the program. The methods of the class is extracted into a CtMethod array. The class file is checked for any kind of synchronization and thread control methods and objects and these are stored in a vector (prMethods) to be processed. The CtMethod array is processed to find, replace and rewrite the bytecodes.

```java
static void processClassFile(ClassPool pool, String cname) {
  Vector prMethods = new Vector();
  Vector<invokeNode> invokeInfos = new Vector<invokeNode>();
  cname = cname.substring(0, cname.indexOf("."));
  try {
    CtClass cc = pool.get(cname);
    boolean interfaceClass = Modifier.isInterface(cc.
        getModifiers());
    boolean abstractClass = Modifier.isAbstract(cc.getModifiers
        ());
    if (!interfaceClass && !abstractClass) {
      // Get the list of methods implemented in the class file
      CtMethod[] ctMethods = cc.getDeclaredMethods();
      // Search the Constant Pool for Methods that needs to be
      // instrumented
      prMethods = processMethods(cname, pool);
      for (int i = 0, n = ctMethods.length; i < n; i++) {
        if (ctMethods[i].getName().equals("main")) {
          instrumentMainStartStop(cname, ctMethods[i].getName(),
              pool);
          System.out
              .println("Control Methods for Main inserted!");
        } else {
          if (prMethods.size() > 0) {
            for (int k = 0; k < prMethods.size(); k++) {
              MethodInf mi = (MethodInf) prMethods
                  .elementAt(k);
              // Get the invocation information the methods in
              // prMethods
              invokeInfos.addAll(getInvoke(mi.cpIndex,
                  mi.name, cname, ctMethods[i].getName(),
                  pool));
            }
            for (int k = invokeInfos.size() - 1; k >= 0; k--) {
              invokeNode inode = (invokeNode) invokeInfos
                  .elementAt(k);
              // Insert the replacement code for the
              // instrumented methods
              replaceCode(inode.methodName, inode.objectName,
                  inode.position, inode.originalOpcode,
                  pool, cname, ctMethods[i].getName());
            }
            System.out
                .println("Finished inserting Controller's
                   substitute codes!\n");
          }
          // Modify the run() method to insert
          // thread control methods for the start/end of the
          // method
          if (ctMethods[i].getName().equals("run")
              && !methodExists(ctMethods, "main")) {
            System.out.println("Instrumenting run() method:");
            instrumentThreadStartStop(cname,
                ctMethods[i].getName(), pool);
          }
          System.out
              .println("Instrumenting Controller's monitor
                 methods");
          instrumentMonitors(cname, ctMethods[i].getName(), pool
              );
          System.out
              .println("\nRemoving replaced methods' bytecode");
          instrumentRemoves(invokeInfos, pool, cname,
              ctMethods[i].getName());
          System.out.println("Checking method access flag");
          // Turn a synchronized method into a normal method
          // Thread Scheduler has the control over sync
          processAccessFlag(cname, ctMethods[i].getName(), pool)
              ;
          if (i + 1 == n)
            cc.writeFile();
          invokeInfos.clear();
        }
      }
    }
  } catch (Exception e) {
    e.printStackTrace();
  }
}
```

Figure 27: ProcessClassFile method.

ProcessMethods method (Figure 28) searches the constant pool for method references. If any of the thread synchronization related methods are found, the method's properties are stored in a plain old java object (POJO) of class `MethodInf` (Figure 29). ProcessClassFile method later searches for these methods for injecting replacement code and removing the original invocations.

getInvoke method finds the invocations of synchronization methods, their parameters and callers. This information is stored in a POJO, called invokeNode (Figure 30), for removal and insertion of related synchronization code. The invokeNode object has the opcode for the method invocation and the index of this invocation in the bytecode.

The bytecode of the processed class' method is searched byte by byte to find a method invocation opcode, when an invocation opcode is found the method called by it is checked with the synchronization method that is being searched for. If there is a match the bytecode is checked to find the object used for synchronization. This method could be called as a bytecode level lexical analyzer and parser. All access types of the methods, all return type of methods and all possible combinations of parameters are considered to find out the correct synchronized object. If the method is synchronized, the synchronization object would be the class itself for `synchronized static` methods. If it is another kind of method, the synchronized object could be

```
/*
 * Search the constant pool for the synchronization methods
 * Store the method info in a vector and return the list
 */
public static Vector processMethods(String cname, ClassPool pool
    ){
  Vector classMethods = new Vector();
  CtClass cc;
  try {
    cc = pool.get(cname);
    ConstPool cp = cc.getClassFile().getConstPool();

    for(int i = 0; i<cp.getSize();i++){
      try{
        // throws exception when current index is not a
            methodref
        String mName = cp.getMethodrefName(i);

        if(mName.equals("notify")||mName.equals("notifyAll")||
            mName.equals("wait")
              ||mName.equals("sleep")||mName.equals("yield")||
                  mName.equals("interrupt")||
              mName.equals("join")){
          MethodInf mi = new MethodInf(i,mName);

          classMethods.add(mi);
          System.out.println(mName);
        }
      }catch(Exception c){}
    }
  }catch(Exception e){
    e.printStackTrace();
  }
  return classMethods;
}
```

Figure 28: ProcessMethods Method

```
class MethodInf{
  public String name;
  public int cpIndex;
  public boolean isSynched = false;

  public MethodInf(int index, String name){
    this.name = name;
    cpIndex = index;
  }
  public MethodInf() {
    name = null;
    cpIndex = -1;
  }
  public String getName() {
    return name;
  }
  public void setName(String name) {
    this.name = name;
  }
  public int getCpIndex() {
    return cpIndex;
  }
  public void setCpIndex(int cpIndex) {
    this.cpIndex = cpIndex;
  }
}
```

Figure 29: MethodInf Class

```
class invokeNode{
  String methodName;
  String objectName;
  int position;
  String originalOpcode;

  public invokeNode(String mName, String oName, int pos, String
      origOpcode){
    this.methodName = mName;
    this.objectName = oName;
    this.position = pos;
    this.originalOpcode = origOpcode;
  }
}
```

Figure 30: InvokeNode Class

a parameter, global variable, local variable or even the returned object of another method call. Every legal expression type is considered to find the correct synchronized object. Finding the synchronized object is important, because it is a parameter of our Thread Scheduler's methods and also there could be multiple synchronized objects in an application. The integrity of the tested application is the most important criteria to uncover the real software errors.

replaceCode method is used to insert the Thread Scheduler version of the synchronization method found in a method. This newly added code is inserted with Javassist's CtMethod class' `insertAt()` method. The Line Number Table of the method is checked for the place original method is called. The Line Number Table holds the program counter for the bytecode of the method. The replacement method is inserted in the bytecode at this program counter.

```
public static void instrumentThreadStartStop(String cname,
    String mname, ClassPool pool){
  CtClass cc;
  try {
    cc = pool.get(cname);
    CtMethod m = cc.getDeclaredMethod(mname);

    // Insert before the method body
    m.insertBefore("{ThreadScheduler.threadHasStarted(this.
        getClass().getSimpleName()+\" , \"+Thread.currentThread()
        .getName()+\" , \"+Thread.currentThread().getStackTrace()
        [1].getLineNumber());}");
    // Insert after the method body
    m.insertAfter("{ThreadScheduler.threadIsAboutToEnd(this.
        getClass().getSimpleName()+\" , \"+Thread.currentThread()
        .getName()+\" , \"+Thread.currentThread().getStackTrace()
        [1].getLineNumber());}");
  } catch (NotFoundException e) {
    e.printStackTrace();
  } catch (CannotCompileException e) {
    e.getCause();
    e.getReason();
  }
}
```

Figure 31: InstrumentThreadStartStop method

InstrumentThreadStartStop method (Figure 31) is used to insert the `threadHasStarted()` and `threadIsAboutToEnd()` methods into the `run()` method. The functionality of these methods were explained in the previous section. Javassist offer an easy way to instrument the start and end points of the code. Instrumenting these parts of the code doesn't require the complex opcode searching methods explained earlier.

InstrumentMonitors method is a specific version of `getInvoke()` method searching for the `monitorenter` and `monitorexit` instructions.

The synchronized object is determined and the replacement code is inserted into the bytecode of the processed method.

Previously explained methods handled the insertion process for the replacement code. **InstrumentRemoves** method is used for removing the method calls for the original methods and all instruction opcodes related to these methods. The removals must the last actions done on a class file because Javassist can produce valid class files only by this order. Otherwise the modified class file could not pass the verification algorithm of the JVM.

The `InstrumentMonitors()` method was used to process the synchronized blocks inside a method. If the method itself is synchronized, the monitor instrumentation should be handled as if it was instrumenting a thread start-stop. The first and last lines of the method is instrumented by the **ProcessAccessFlag** method (Figure 32), with the Thread Scheduler's monitor methods. For synchronized methods, access flag must be altered to normal method flag, since the Thread Scheduler now has the control with the instrumented methods.

```
public static void processAccessFlag(String cname, String mname,
    ClassPool pool){
  CtMethod method;
  try {
    method = pool.getMethod(cname, mname);
    MethodInfo minfo = method.getMethodInfo();
    int accessFlag = minfo.getAccessFlags();
    int synch = accessFlag & Integer.parseInt("100000", 2);
    int staticMethod = accessFlag & Integer.parseInt("1000", 2);
    if(synch == 32){
      if(staticMethod == 8){
        method.insertBefore("{ThreadScheduler.myMonitorEnter("+
            cname+".class, this.getClass().getSimpleName()+\" ,
            \"+Thread.currentThread().getName()+\" , \"+Thread.
            currentThread().getStackTrace()[1].getLineNumber());}
            ");
        method.insertAfter("{ThreadScheduler.myMonitorExit("+
            cname+".class, this.getClass().getSimpleName()+\" ,
            \"+Thread.currentThread().getName()+\" , \"+Thread.
            currentThread().getStackTrace()[1].getLineNumber());}
            ");
      }
      else{
        method.insertBefore("{ThreadScheduler.myMonitorEnter( $0
            , this.getClass().getSimpleName()+\" , \"+Thread.
            currentThread().getName()+\" , \"+Thread.
            currentThread().getStackTrace()[1].getLineNumber());}
            ");
        method.insertAfter("{ThreadScheduler.myMonitorExit($0,
            this.getClass().getSimpleName()+\" , \"+Thread.
            currentThread().getName()+\" , \"+Thread.
            currentThread().getStackTrace()[1].getLineNumber());}
            ");
      }
      // Remove the synchronized bit
      minfo.setAccessFlags(accessFlag-32);
    }
  } catch (NotFoundException e) {
    e.printStackTrace();
  } catch (CannotCompileException e) {
    e.printStackTrace();
  }
}
```

Figure 32: ProcessAccessFlag Method

4.5 Caveats

The Thread Scheduler aims to acquire full control of the concurrently running threads from the JVM. Instrumenting the class files with the provided methods makes this objective achievable. But the Thread Scheduler's and the Tester's accuracy highly depends on the JVMs processing of the main method and initialization of threads. If the Thread Scheduler is started by the main method of the testing program very early on the initialization process. Some of the threads might get blocked from running which may cause abrupt stops of the Testing System. Therefore, even if its `startTesting()` method is called prematurely, the Thread Scheduler must suspend its execution for a short time to let each thread to register itself to the system. This is required because of the Tester Algorithm. The first run of the tester is an exploratory run which tries to identify as many distinct blocks (thread interleavings) as possible.

For purposes of simplicity, our test applications are assumed to have their main methods in a certain format. The main() method initializes the threads that will run concurrently, calls the start() methods to run the threads. After initializing and starting every thread, the main method alerts the Thread Scheduler to start testing and closes. The problem mentioned in the previous paragraph seems to be related with this assumption. It is related, but to a lesser degree. The previous problem is due to the inner mechanics of the JVM to initialize and run the threads. This assumption is made to reduce the complexity of the test system and the possibility of deadlocks depending on synchronization in the main thread even before the control of the system is handed over to the Thread Scheduler.

The Javassist BCI library had some restrictions on the availability of some operations on the bytecode of a class file. The library allowed us to use global variables and parameters, but this was not enough to instrument a class file with our code. The programs may deal with parameters, function calls and local variables as their synchronized variables. The Javassist Library does not allow access to local variables declared in the processed method. We have extended the Javassist library to allow accessing these variables. We need the target application to be compiled with a special argument "-g" to access the local variable tables and finding out the object that we are looking for. The compilation argument "-g" means keeping the debugging information in the class file. We have also implemented the reverse engineering methods to build the source code for a method with arguments. This source code would later be used as an argument for our Thread Scheduler's methods. This is especially useful if a reverse engineered method's returned object is the synchronized object (See figure 33).

```
......
{
  int index = 1;
  synchronized( objectArrayList.get(index) ){
     ......
  }
   ......
}
......
```

Listing 1: Original Code Segment

```
......
{
  int index = 1;
  ThreadScheduler.myMonitorEnter( objectArrayList.get(index) );
     ......
  ThreadScheduler.myMonitorExit( objectArrayList.get(index) );
   ......
}
......
```

Listing 2: Instrumented Code Segment

Figure 33: Reverse Engineered method in an instrumented class file. The get method, the class object (objectArrayList) it is called from and the parameter list (index) is reverse engineered to form the correct statement and used as parameter of myMonitorEnter and myMonitorExit methods.

5 Interleaving Coverage Criteria Oriented Testing of Multithreaded Applications

The concurrency related errors have been researched for some time. Data race, deadlock and atomicity bug topics have been intensely researched and are easier to test with conventional testing techniques. Our research focuses on the **Ordering Errors**, which is directly affected by the scheduling of the concurrent threads. Catching ordering related bugs require testing every possible ordering of thread schedules. But exhaustive testing for all possible schedules requires extensive amount of tests and almost impossible to cover.

Our approach is dividing the concurrent threads into smaller execution units, called atomic blocks. Atomic blocks have been explained in the previous section. These atomic blocks generates a graph of blocks, showing the thread interleavings, Katayama et. al [10] calls them Event Interactions Graph. This graph is dynamically updated as the tests continue and new atomic blocks are found. Figure 34 is an example of a Thread with 6 blocks. The figure shows that the the thread has a starting block numbered 0, we can say that 0 is the starting atomic block of this thread, unless our Tester finds a new Block that starts from the same line but finishes in a different place (Ex: 35). First interleaving figure also has two looping blocks. A loop complicates our testing system since it allows more interleaving sequences to be legal, also increasing the number of tests to reach full coverage.

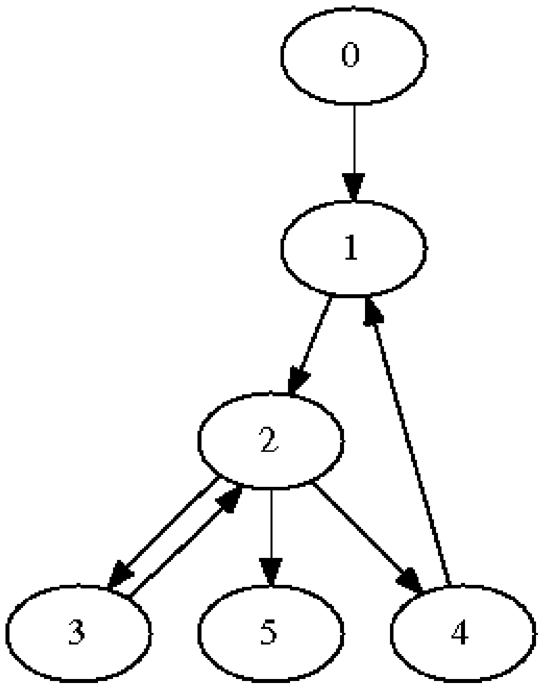


Figure 34: A thread interleaving graph showing the execution order of atomic blocks.

Interleaving Coverage Criteria is based on dividing a thread's execution into smaller units and checking a batch of very small subsequences of interleavings for the coverage of exhaustive schedules. These subsequences are based on Sequence Covering Arrays. Even a very small number of atomic blocks would lead to an unmanageable number of schedules. A full schedule to finish a test covers a big number of these subsequences (Figure 36). We have defined the length of these subsequences as t and they are called t-way sequences. Our tests have been done with sequence lengths of 2, 3 and 4.

The Interleaving Coverage Criteria Oriented Tester is a hybrid testing system. The tester tries to increase the t-way sequence coverage, while trying

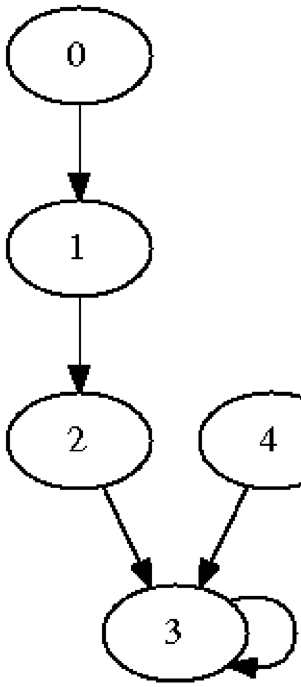


Figure 35: Another thread interleaving graph showing the execution order of atomic blocks. The atomic blocks 0 and 4 are two distinct blocks that starts the thread execution.

to uncover ordering related bugs. The tester algorithm starts with a random sweep of the system, uncovering as much atomic blocks as possible. The tester schedules the registered threads randomly, this approach is similar to what Stoller proposed in [9]. After each scheduling choice a new atomic block is found. These blocks are added to their thread's interleaving tree, with the information of the previous thread. The thread interleaving tree is required to filter out the sequences that would be impossible to reach. This is a performance increasing measure. The tester knows which way to go

```
N = 6 : # of atomic blocks
t = 3

Test 1 Schedule = [1, 2, 3, 4, 5, 6]
# of Exhaustive Tests: 6! = 720
# of Interleaving Coverage tests = 120
# of Covered t-way sequences with 1 test = 10
[1, 2, 3] [2, 3, 4] [3, 4, 6]
[1, 2, 4] [2, 3, 5] [4, 5, 6]
[1, 2, 5] [2, 3, 6]
[1, 2, 6] [3, 4, 5]
```

Figure 36: The expected number of exhaustive tests versus the expected number of t-way sequence interleaving coverage tests. The number of t-way sequences covered with only one test shows the efficiency of this method.

with and leaving out impossible schedules reduces the number of tests and sequences to be covered.

The second test type implemented into the system is Thread Sequence Covering Array (TSCA) testing. We have previously mentioned employing sequence covering arrays for thread interleavings. The TSCA testing approach applies this concept to make an abstraction for scheduling the threads. The tester already has a *t* value assigned and we prepare predefined schedules of threads with the t-way sequence covering rule. To schedule a thread with a TSCA, the entries of the TSCA are checked for availability and the matching thread which is available to be scheduled is chosen. when the first entry is not available at any time the next entry is checked for availability and this continues until a matching pair of TSCA choice and available thread is found. Applying the TSCA testing algorithm significantly increased the coverage ratio of our tests.

The last testing type is Temporary Sequence Covering Array (TempSCA) testing. All previous scheduling information is stored by the tester. After the random testing and TSCA testing is finished, if there are uncovered t-way schedules left, the Tester selects t-way sequences and finds the schedules that leads to each atomic block in the selected t-way sequence. These distinct schedules that reach these atomic blocks are then combined into a single schedule. TempSCA testing uses this schedule to test the application. This approach tries to increase the coverage ratio of t-way sequences as much as possible.

5.1 Thread Interleaving Coverage Analysis

Ordering related errors are likely to occur because of the programmer's assumptions on the behaviour of the system and the thread handling strategy of the JVM. Catching ordering related errors is harder because the JVM does not guarantee behaving and controlling the schedule the way it does during error-free executions of the application. The thread scheduling may change due to the load of the system or a rare input to the system may lead to an unexpected code segment to execute.

Sequence Covering Arrays offer an efficient way of testing the execution segments. In a schedule prepared by our tester, we can assume that any t-1 successors of a block in a test schedule is covered by our system. In the exhaustive testing environment, a tester would need to schedule each of these blocks immediately after a selected block in order to count these segments covered. However when we think about the threads in the application, a threads execution may never affect another thread or on the contrary these threads would be highly coupled. Our approach is based on this exclusiveness of the blocks.

5.1.1 Exhaustive Testing

A concurrent program that has m threads, each having n atomic blocks. Equation (1) can be used to calculate the number of exhaustive schedules. For a fairly small example of $m = 5$ and $n = 5$, the number of schedules becomes 623,360,743,125,120. The total number of schedules makes it almost impossible to test, even if we assumed a test to take 1 microsecond to complete, we would need 173,000 hours to test.

$$\prod_{i=0}^{m-1} \binom{mn - in}{n} \tag{1}$$

5.1.2 Coverage Criteria Oriented Testing

In this testing method, we assure that any combination of t-way sequences are tested. The number of sequences are becomes larger as the number of blocks but they are still significantly smaller than an exhaustive test. Another point is that these t-way sequences are compressed into thread schedules and even one test of random sweep would cover a large number of t-way sequences.

$$0 \leq a_i \leq t$$
$$0 \leq a_i$$
$$M = m * n$$
$$\sum_{a_1+a_2+\ldots+a_m=t} \prod_{i=1}^{M} \sum_{k=1}^{n-a_i+i} \left[\binom{n-k}{a_i-1} \binom{t - \sum_{j=1}^{i-1} aj}{a_i} \right]$$

Figure 37: The formula of calculating the number of thread t-way subsequences.

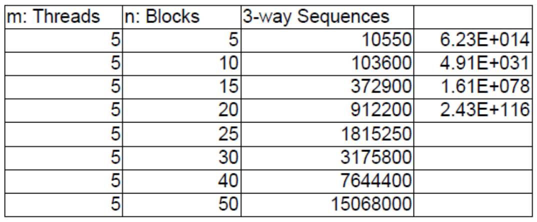
The test results will be given in the next chapter. If we take the example from the Exhaustive Testing section, m threads, each having n atomic blocks, the number of t-way sequences can be calculated with the following equation in Figure 37. For $m = 5$, $n = 5$ and $t = 3$, the number of t-way sequences becomes 10,550. The number of tests to reach full coverage becomes only 63.

6 Case Studies

The Interleaving Coverage Criteria Oriented Test System have been tested with various sample concurrent applications. A project was used for benchmarking purposes to show the strength of our proposed system againts the exhaustive testing. As we have previously mentioned, an application that contains m threads and n atomic blocks per thread, exhaustive testing has a boundary of mn^{mn}. Figure 38 clearly shows the ineffectiveness of the exhaustive testing for concurrent systems. Our Sequence Covering Array approach remains scalable even when the number of blocks rise rapidly. The execution times for each test remains constant because of the efficiency of the thread scheduling algorithm. Only time consuming action done by the system is generating the t-way sequences for larger numbers. This is a one time process and its space complexity is inversely proportional to time. Only the uncovered sequences are stored in the data structure.

6.1 Benchmarking Application

The benchmarking application (Figure 39) is a straightforward program. The class extends the `Thread` class and has a combination of critical and non-critical sections. Our algorithm guarantee high coverage levels for each value of sequence length t. For t = 2, our system reaches full coverage in just 3 tests for any number of m and n (Table 3).



m: Threads	n: Blocks	3-way Sequences	
5	5	10550	6.23E+014
5	10	103600	4.91E+031
5	15	372900	1.61E+078
5	20	912200	2.43E+116
5	25	1815250	
5	30	3175800	
5	40	7644400	
5	50	15068000	

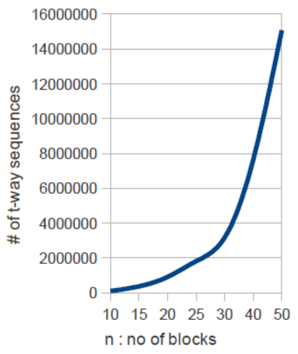


Figure 38: The number of t-way sequences for various values of m and n and respective Exhaustive Testing tests.

m: # of Threads	n: # of Blocks	t: length	# of Tests	Interleaving Coverage
5	5	2	3	100 %
10	5	2	3	100 %
15	5	2	3	100 %
20	5	2	3	100 %
25	5	2	3	100 %
30	5	2	3	100 %
40	5	2	3	100 %
50	5	2	3	100 %

Table 3: Results of benchmarking tests with t = 2.

```java
public class Single extends Thread{

  private int stop = 5;
  SharedObject lock;
  SharedObject lock2;

  public Single(SharedObject lock, SharedObject lock2){
    this.lock = lock;
    this.lock2 = lock2;
  }

  public void run() {

    System.out.println("\nSingle: ( "+this.getName()+" ) Non-
        Critical Section 1, value:"+lock.toString());

    synchronized(lock){
      lock.update();
      System.out.println("\nSingle: ( "+this.getName()+" )
          Critical Section 1, value:"+lock.toString());
    }

    System.out.println("\nSingle: ( "+this.getName()+" ) Non-
        Critical Section 2, value:"+lock.toString());

    synchronized(lock){
      lock.update();
      System.out.println("\nSingle: ( "+this.getName()+" )
          Critical Section 2, value:"+lock.toString());
    }

    System.out.println("\nSingle: ( "+this.getName()+" ) Non-
        Critical Section 3, value:"+lock.toString());

    synchronized(lock2){
      lock.update();
      System.out.println("\nSingle: ( "+this.getName()+" )
          Critical Section 3, value:"+lock2.toString());
    }

    System.out.println("\nSingle: ( "+this.getName()+" ) Non-
        Critical Section 4, value:"+lock.toString());

    synchronized(lock){
      lock.update();
      System.out.println("\nSingle: ( "+this.getName()+" )
          Critical Section 4, value:"+lock.toString());
    }
  }
}
```

Listing 3: Original Code Segment

m: # of Threads	n: # of Blocks	t: length	TSCA tests	Coverage TSCA	Greedy Tests	Coverage %	Total Tests	# of Exhaustive Tests
5	5	3	8	93.47%	61	100%	70	$6.23x10^{14}$
10	5	3	14	96.69%	146	100%	161	$4.91x10^{31}$
15	5	3	20	94.46%	379	99.67%	400	$1.61x10^{78}$
20	5	3	22	94.48%	546	96.97%	569	$2.43x10^{116}$
25	5	3	26	97.14%	86	98.87%	113	$1.97x10^{157}$
30	5	3	28	96.19%	15	96.87%	44	$2.4x10^{200}$
40	5	3	32	96.61%	12	96.95%	45	$5x10^{291}$
50	5	3	34	96.02%	3	96.29%	38	$3.55x10^{388}$

Table 4: Results of benchmarking tests with t = 3. TSCA Algorithm significantly increases the coverage ratio, while keeping the required tests at minimum.

m: # of Threads	n: # of Blocks	t: length	# of Tests	Interleaving Coverage
5	5	2	3	100 %
5	5	3	8	93.47 %
5	5	4	28	72.68 %

Table 5: Results of benchmarking tests with m = 5, n= 5 and t = 2,3,4 for TSCA tests.

TSCA testing is very important for increasing the coverage percentage of the system. For $t = 3$, the influence of the TSCA is given in the Table 6.1. The number of tests to reach 100 % coverage takes more tests than $t = 2$. The comparison of $t = 2,3,4$ is given in Table 5.

The information in tables 3 and would be misleading. In these tables a t value of 2 seems enough to reach maximum interleaving coverage. The TSCA algorithm is a divide and conquer type of algorithm and bigger sequence lengths are closer to the real problem and they produce better simulation of the real system. So we have a trade-off situation, increase the length of sequences to reach more accurate behaviour, while risking exponential growth on number of tests or decreasing the sequence length to predict the behavior from a narrower view of the system.

6.2 Real Applications

The Interleaving Coverage Criteria Oriented Tester has been tested on real concurrency application examples from the University of Nevada Lincoln's Software-artifact Infrastructure Repository (SIR). The examples that are suitable for testing according to our assumption have been tested. Figures **??** and **??** are examples of these applications. We have tested for ordering related errors and while our Tester was trying to explore every legal interleaving combination, We have observed this errors in these programs. The applications in these figures have strictly coupled threads that have to execute in some particular order. Our Tester forces the program out of this scheduling pattern to uncover the ordering errors. Figure 6 show the coverage ratio of the system as the tests are administered.

The Clean Project of the SIR

```
public class Main {
    static int iFirstTask=1;
    static int iSecondTask=1;
    static int iterations=12;

    public static void main(String[] args) {
      if (args.length < 3){
  System.out.println("ERROR: Expected 3 parameters");
      }else{
  iFirstTask = Integer.parseInt(args[0]);
  iSecondTask = Integer.parseInt(args[1]);
  iterations = Integer.parseInt(args[2]);
  Main t= new Main();
  System.out.println("Starting...");
  //t.run();
  Event new_event1 = new Event();
        Event new_event2 = new Event();
        System.out.println("fjhgj");
  for (int i=0;i<iFirstTask;i++)
     new FirstTask(new_event1,new_event2,iterations).start();
  for (int i=0;i<iSecondTask;i++)
     new SecondTask(new_event1,new_event2,iterations).start();
      }
   }
}
```

Listing 4: Main.java

```
public class Event {
    public int count = 0;

    public synchronized void waitForEvent(int remote_count) {

  if (remote_count == count)
    try {
      wait();

    }catch(InterruptedException e) {};
    }

    public synchronized void signal_event() {
        count = (count + 1) % 100;
  System.out.println("signal_event() count: "+count);
        notifyAll();
    }
}
```

Listing 5: Event.java

```
public class FirstTask extends Thread {
    int iterations;
    Event event1,event2;

    public FirstTask(Event e1, Event e2,int iterations) {
  this.event1 = e1;
  this.event2 = e2;
  this.iterations=iterations;
    }

    public void run() {
  System.out.println("First task start");
  int count = 0;
        count = event1.count;
        for (int i=0;i<iterations;i++) {
    System.out.println("1: event1.waitForEvent();");
        event1.waitForEvent(count);
        count = event1.count;
    System.out.println("1: Count: "+count+ " i: "+i);
    System.out.println("1: event2.signal_event();");
        event2.signal_event();
        }
  System.out.println("First task stop");
    }
}
```

Listing 6: FirstTask.java

```
public class SecondTask extends Thread {
    int iterations;
    Event event1,event2;

    public SecondTask(Event e1, Event e2,int iterations) {
        this.event1 = e1;
        this.event2 = e2;
        this.iterations=iterations;
    }

    public void run() {
  System.out.println("Second task start");
        int count = 0;
        count = event2.count;
        for (int i=0;i<iterations;i++) {
    System.out.println("2: event1.signal_event();");
        event1.signal_event();
    System.out.println("2: event2.waitForEvent(count);");
        event2.waitForEvent(count);
        count = event2.count;
    System.out.println("2: Count: "+count+ " i: "+i);
        }
  System.out.println("Second task stop");
    }
}
```

Listing 7: SecondTask.java

The TwoStage Project of the SIR

```
public class Main{
    static int iTthreads=1;
    static int iRthreads=1;
    static TwoStage ts;
    static Data data1,data2;

  public static void main(String[] args) {
    if (args.length < 2){
      System.out.println("ERROR: Expected 2 parameters");
    }else{
      iTthreads = Integer.parseInt(args[0]);
      iRthreads = Integer.parseInt(args[1]);
      data1=new Data();
      data2=new Data();
      ts = new TwoStage(data1,data2);
      for (int i=0;i<iTthreads;i++)
        new TwoStageThread(ts).start();
      for (int i=0;i<iRthreads;i++)
        new ReadThread(ts).start();
    }
  }
}
```

Listing 8: Main.java

```
public class Data {
    public int value;
    public Data() {
      value=0;
    }
}
```

Listing 9: Data.java

```
public class ReadThread extends Thread {
  TwoStage ts;
  public ReadThread(TwoStage ts) {
     this.ts=ts;
  }

  public void run() {
     ts.B();
  }
}
```

Listing 10: ReadThread.java

```
public class TwoStage {

      public Data data1,data2;

      public TwoStage (Data data1, Data data2) {
   this.data1=data1;
   this.data2=data2;
      }

      /*
       * This method is used to simulate two stage access
       * In first stage, it modify the value of data1
       * In the second stage, it modify the value of data2
           according to data1
       * It assumes that data2.value=data1.value (This assumption
           is used to
       * simulate in some database application, the two values in
           different
```

```
     * tables must be consistant.
     */
    public void A () {

        // This is the first stage
        synchronized (data1) {
            data1.value=1;
        }

        //reading may happen here, data will be inconsistant...

        // This is the second stage, using the result of stage1
            to calculate
        synchronized (data2) {
            //if other threads modify data1.value, inconsistancy
                will happen
            data2.value=data1.value+1;
        }
    }

    public void B () {

  int t1=-1, t2=-1;
          synchronized (data1) {
       if (data1.value==0) return ; //The first stage has not
           begun.
            t1=data1.value;
```

```
      System.out.println("data1: "+data1.value+" was modified
          elsewhere, t1: "+t1);
   }
   synchronized (data2) {
        t2=data2.value;
   System.out.println("data2: "+data2.value+", t2: "+t2);
   }
          if (t2 != (t1+1))
        throw new RuntimeException("bug found");
      }
}
```

Listing 11: TwoStage.java

```
public class TwoStageThread extends Thread {
   TwoStage ts;
   public TwoStageThread(TwoStage ts) {
      this.ts=ts;
   }

   public void run() {
      ts.A();
   }

}
```

Listing 12: TwoStageThread.java

Test #	Total Sequences	Covered Sequences	Coverage %	Covered this run
1	9928	6090	61.34 %	6090
2	9928	7479	75.33 %	1389
3	9928	8005	80.63 %	526
4	9928	8260	83.19 %	255
5	9928	8268	83.27 %	8
6	9928	8285	83.45 %	17
7	9928	8288	83.48 %	3
8	9928	8296	83.56 %	8
9	9928	8302	83.62 %	6
10	9928	8305	83.65 %	3
11	9928	8474	85.35 %	169
12	9928	8476	85.37 %	2
13	9928	8480	85.41 %	4
14	9928	8490	85.51 %	10
15	9928	8509	85.70 %	19
16	9928	8518	85.79 %	9
17	9928	8521	85.82 %	3
18	9928	8531	85.92 %	10
19	9928	8536	85.97 %	5
20	9928	8537	85.98 %	1
21	9928	8545	86.06 %	8

Table 6: The Coverage information after each test of Clean Project. t = 3

Test #	Total Sequences	Covered Sequences	Coverage %	Covered this run
1	5550	4734	85.29%	4734
2	5806	4990	85.94%	256
3	10540	10177	96.55%	5187
4	10540	10202	96.79%	25
5	10540	10382	98.50%	180
6	10540	10531	99.91%	149
7	10540	10532	99.92%	1
8	10540	10540	100%	8

Table 7: The Coverage information after each test of DiningPhilosophers Project. t = 3. The table shows that after test # 3 new blocks are found and new sequences are added to our coverage

Project	m	n	t	TSCA tests	Coverage TSCA	Greedy Tests	Coverage %	Total Tests	# of Exhaustive Tests
Clean	4	6	3	8	61.34%	48	86.06%	57	2,3x10^{12}
Bounded Buffer	4	7	3	8	11.34%	2415	70.29%	2424	$9,3x10^{10}$
Dining Philosophers	5	4	3	8	85.29%	6	100%	15	$3x10^{11}$
Two Stage	4	4	3	-*	-*	894	89.56%	895	$2,6x10^{6}$

Table 8: The testing information about the real applications, t = 3. (*) TSCA method can not be used and coverage ratio for TSCA tests does not exist. Only random exploration and greedy algorithms are used for testing.

7 Concluding Remarks

Concurrency is one of the biggest pitfalls in today's Software Engineering practice. Our everyday lives depend on applications that has to serve a large number of the people at the same time. Concurrent programs are prone to software errors and these errors are harder to spot. Therefore testing a concurrent application requires efficient programs that are able to expose weaknesses of an application. The best way to find all weaknesses is exhaustive testing but it is practically impossible to exhaustively test a relatively small program.

In this thesis, we envision a test process that aims to obtain a full coverage under a given thread interleaving coverage criterion. The coverage criterion implicitly defines the space of interesting thread interleavings for testing. Given a coverage criterion, our ultimate goal is to cover all required inter-

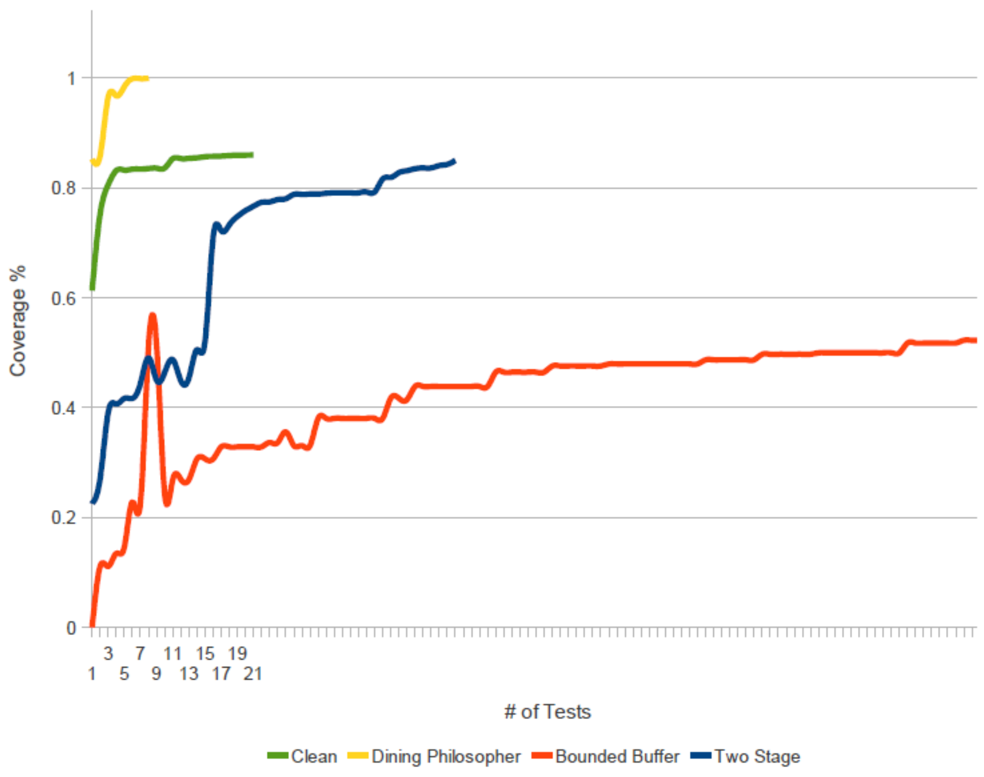


Figure 40: The comparison of real applications coverage ratios.

leavings in a minimum number of thread schedules, thus to reduce the cost of testing. In this work, we however addressed only a practical instantiation of this vision. Our coverage criterion was to cover all t-way combinations of atomic blocks (not necessarily to be consecutive), so that all ordering errors caused by the interaction of t or less number of atomic blocks can reliably be revealed. To evaluate the proposed approach, we conducted a series of experiments. Although carried out in a small scale, the results of our experiments suggest that 1) using t-way sequence coverage criterion is an effective way of revealing ordering errors and 2) the proposed approach can obtain reasonable coverage at a fraction of the cost compared to exhaustive testing.

8 Future Work

Our project focuses on the applicability of the Sequence Covering Arrays to concurrency bug checking, especially ordering errors. The need to implement such an approach was a necessity to produce an efficient way of testing instead of the extremely costly exhaustive search. The Tester program proved to be useful in reducing the number of tests and testing the ordering related errors. The Tester system is a hybrid system and a future direction for the project would be increasing the effectiveness of the TempSCA approach to produce high density coverage schedules. If we are to produce a better block reachability based testing scheduler, we need to know the interaction between threads, that we do not know in our current approach. This is not an easy extension to go with, because it would require through code analysis to find the dependencies of threads. The statement to statement code analysis would require a more complex Instrumenter/Analyzer tool.

The second improvement to our system would be increasing the types of coverage supported by our Tester. Currently we are supporting Sequence Covering Arrays and thread interleavings depending on these sequences. This approach is efficient for testing ordering errors, but implementing another coverage criteria may improve the systems performance and lets the tester cover more types of concurrent software bugs and bug patterns.

References

[1] Scott Oaks, Henry Wong, *Java Threads.* O'Reilly, 3rd Edition, 2004.

[2] Joshua Engel, *Programming for the Java Virtual Machine.* Addison Wesley, 1999.

[3] Shan Lu, Weihang Jiang, Yuanyuan Zhou, *A Study of Interleaving Coverage Criteria.* ESEC/FSE, 2007.

[4] Shan Lu, Soyeon Park, Eunsoo Seo, Yuanyuan Zhou, *Learning from Mistakes A Comprehensive Study on Real World Concurrency Bug Characteristics.* ASPLOS, 2008.

[5] Eitan Farchi, Yarden Nir, Shmuel Ur, *Concurrent Bug Patterns and How to Test Them.* Parallel and Distributed Processing Symposium, 2003.

[6] Marcel Christian Baur, *Instrumenting Java Bytecode to Replay Execution Traces of Multithreaded Programs.* Diploma Thesis, 2003.

[7] Derek L. Bruening, *Systematic Testing of Multithreaded Java Programs.* Master's Thesis, 1999.

[8] Richard Bounds, *Virtual Time Execution.* Master's Thesis, 2009.

[9] Scott D. Stoller, *Testing Concurrent Java Programs using Randomized Scheduling.* Proc. Second Workshop on Runtime Verification (RV), 2002.

[10] Tetsuro Katayama, Eisuke Itoh, Zengo Furukawa, *Test-case Generation for Concurrent Programs with the Testing Criteria Using Interaction Sequences*. APSEC, 1999.

[11] Gwan-Hwan Hwang, Kuo-Chung Tai, Ting-Lu Huang, *Reachability Testing: An Approach to Testing Concurrent Software*. International Journal of Software Engineering and Knowledge Engineering, 1995.

[12] Yu Lei, Richard Carver, *A New Algorithm for Reachability Testing of Concurrent Programs*. ISSRE, 2005.

[13] Cormac Flanagan, Stephen N. Freund, *Atomizer: A Dynamic Atomicity Checker For Multithreaded Programs*. POPL, 2004.

[14] David Coppit, Jinlin Yang, Sarfraz Khurshid, Wei Le, Kevin Sullivan, *Software Assurance by Bounded Exhaustive Testing*. International symposium on Software testing and analysis, 2004.

[15] D. Richard Kuhn, Vadim Okun, *Pseudo-Exhaustive Testing for Software*. Proceedings of the 30th Annual IEEE/NASA Software Engineering Workshop, 2006.

[16] Yaniv Eytani, Klaus Havelund, Scott D. Stoller, Shmuel Ur, *Toward a Framework and Benchmark for Testing Tools for Multi-Threaded Programs*. Conc. & Comp.: Practice & Experience Vol 19, 2007.

[17] Madanlal Musuvathi, Shaz Qadeer, Thomas Ball, Gerard Basler, *Finding and Reproducing Heisenbugs in Concurrent Programs*. Proceedings

of the 8th USENIX conference on Operating systems design and implementation, 2008.

[18] Madan Musuvathi, Shaz Qadeer, *Iterative Context Bounding for Systematic Testing of Multithreaded Programs.* PLDI, 2007.

[19] Stefan Savage, Michael Burrows, Greg Nelson, Patrick Sobalvarro, Thomas Anderson, *Eraser: A Dynamic Data Race Detector for Multithreaded Programs.* ACM Transactions on Computer Systems Vol 15, 1997.

[20] Rajagopalan Sirinivasan, Sandeep K. Gupta, Melvin Breuer, *An Efficient Partitioning Strategy for Pseudo-Exhaustive Testing.* 30th ACM/IEEE Design Automation Conference, 1993.

[21] Liqiang Wang, Scott D. Stoller, *Runtime Analysis of Atomicity for Multithreaded Programs.* IEEE Transactions on Software Engineering, 2006.

[22] Jacob Burnim, Koushik Sen, *Asserting and Checking Determinism for Multithreaded Programs.* Proceedings of the the 7th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering, 2009.

[23] William Perry, *Effective methods for software testing.* John Wiley & Sons, 3rd Edition, 2006.

[24] Zhifeng Lai, S. C. Cheung, *Detecting Atomic-Set Serializability Violations in Multi-threaded Programs through Active Randomized Testing.*

Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering, 2010.

[25] Koushik Sen, *Race Directed Random Testing of Concurrent Programs*. Proceedings of the ACM SIGPLAN conference on Programming language design and implementation, 2008.

[26] Thomas Gschwind, Johann Oberleitner, *Improving Dynamic Data Analysis with Aspect-Oriented Programming*. Proc. Software Maintenance and Reengineering, 2003.

[27] Jan Tretmans, *Testing Concurrent Systems: A Formak Approach*. CONCUR, 1999.