

The Location Pairs Coverage Metric for Testing of Concurrent Programs

by

M. Erkan Keremoğlu

A Thesis Submitted to the
Graduate School of Engineering
in Partial Fulfillment of the Requirements for
the Degree of

Master of Science

in

Electrical & Computer Engineering

Koç University

August, 2007

Koç University
Graduate School of Sciences and Engineering

This is to certify that I have examined this copy of a master's thesis by

M. Erkan Keremoğlu

and have found that it is complete and satisfactory in all respects,
and that any and all revisions required by the final
examining committee have been made.

Committee Members:

Asst. Prof. Serdar Taşiran (Advisor)

Asst. Prof. Öznur Özkasap

Asst. Prof. Ali Sezgin (Yeditepe University)

Date: _____

To my parents and Ercan

“Anneme, babama ve Ercan’a”

ABSTRACT

In this thesis, we present a coverage metric, namely the Location Pairs (LP) coverage, for improving the testing of programs which have concurrently running components. The LP coverage metric is inspired by an execution pattern, encountered in practically used software, that leads to high-level concurrency errors. A high-level concurrency error occurs if a set of thread-shared variables are needed to be accessed atomically, but some threads access them non-atomically. These errors are different from low-level data races, because to avoid data races, protecting single variables is sufficient. The LP coverage metric aims to detect if a pattern of two consecutive actions is executed.

The measurement of the LP coverage metric is implemented using a Java virtual machine and a static race detection tool. First, we detect accesses to shared-variables using static analysis of the race detection tool. Second, location pairs are extracted using this information. Then, we monitor the execution of the tests and record any consequent accesses to locations which accesses the same shared data. At the end, we report covered and uncovered pairs to the developer. The developer can create more interesting scenarios using coverage report and can potentially prevent high-level concurrency errors by testing rarely executed scenarios. We run our tool on several Java benchmarks to show that the tool can practically be used on large-scale programs and the LP coverage metric helps better to find concurrency errors compared to existing coverage techniques for sequential programs.

ÖZETÇE

Bu tezde kořut zamanlı alıřan programların test programı kapsam ölümlerini yaparak, kullanıcının test programını geliřtirmesi için kullanılabilcek konum çiftleri (Location Pairs) ölütü sunulmaktadır. Location Pairs (LP) ölütü akademi ve endüstride kullanılan kořut zamanlı alıřan programlarda sıklıkla karşılaşılan bir yüksek düzey hata örüntüsünden esinlenerek yaratılmıştır. Yüksek düzey hatalar kořut zamanlı alıřan bir programın bileřenlerinin arasında paylaşılan ve atomik olarak deėiřtirilmesi gereken deėiřkenlerin atomik olarak deėiřtirilmemesi sonucu ortaya ıkar. Bu tür hatalar alt düzey hatalardan farklıdır ünkü bu tür hataları önlemek için paylaşılan deėiřkenlerin tek tek korunması yerine birden fazla deėiřken birlikte korunmalıdır.

LP ölütü bir Java sanal makinesi ve statik yarış tespit aracı kullanılarak kodlanmıştır. İlk olarak statik yarış tespit aracı kullanılarak paylaşılan deėiřkenler tespit edilmektedir. İkinci adımda konum çiftleri paylaşılan deėiřkenler taranarak tespit edilmektedir. Daha sonra test programının yürütümü testin alıřtırılması esnasında sanal makine aracılığı ile gözlemlenmektedir. Son adımda gözlemler sonucu elde edilen ölüm sonuçları kullanıcıya rapor edilmektedir. Kullanıcı bu sonuçlara göre daha fazla konum çiftini kapsayacak test senaryoları yaratabilir. Yaptığımız deneylerde de gösterdiğimiz gibi kořut zamanlı alıřan programların testlerinde LP ölümü yüksek düzey hataların bulanmasında kullanıcılara yardımcı olmaktadır.

ACKNOWLEDGMENTS

I would like to thank to my supervisor Asst. Prof. Serdar Taşiran. His patience, understanding and teaching abilities helped me a lot during my graduate studies.

I would like to thank the members of my thesis committee for their valuable comments.

I am grateful to my parents, Osman and Fatma Keremoğlu and my brother Ercan for their endless support and patience.

I am also grateful to Tayfun who helped me a lot whenever I had difficulties during my studies and also for being a great friend and colleague.

This work is supported by TUBITAK.

I would like to thank Ekin Akkuş and Rachel Chan not only for proof-reading of this thesis and for their valuable comments but also being wonderful friends.

TABLE OF CONTENTS

List of Tables	x
List of Figures	xi
Nomenclature	xiii
Chapter 1: Introduction	1
1.1 Goals and Motivation	1
1.2 Contributions	5
1.3 Organization	7
Chapter 2: Concurrency and Concurrency Bugs	8
2.1 Correctness for Concurrent Software	8
2.1.1 General Concurrency Concepts	8
2.1.2 Concurrency in Java	9
2.1.3 Concurrency Problems	13
2.2 Concurrency Bugs	14
2.2.1 Low Level Data Races	14
2.2.2 High Level Data Races	17
2.3 Motivating Example	21
2.4 Verification Techniques	23
2.4.1 Static Techniques	23
2.4.2 Dynamic Techniques	28
2.5 Focusing On Developer’s Intent	33

Chapter 3:	Testing and Coverage Metrics	36
3.1	What is Coverage Used For?	36
3.2	Coverage Metrics for Sequential Programs and Concurrency	37
3.2.1	Coverage Metrics for Sequential Programs	37
3.2.2	Testing Concurrent Programs is Different	39
3.3	Existing Coverage Methods for Concurrent Programs	40
Chapter 4:	The LP Coverage Metric	46
4.1	Concurrent Programs: Syntax and Semantics	46
4.1.1	Syntax and Semantics	46
4.1.2	Locations and Transitions in Java	48
4.2	The Aim of the LP Coverage Metric	50
4.3	Definition of the Metric	52
4.3.1	The Coverable Pairs Set	52
4.3.2	Measuring Coverage	53
4.3.3	Reporting Coverage	54
4.4	The Implementation	55
4.4.1	Static Analysis with Chord	55
4.4.2	Runtime-monitoring with JPF	60
4.4.3	Reporting Coverage	64
Chapter 5:	Experimental Results	65
5.1	Experiments on Apache FTPServer	65
5.1.1	Apache FTPServer	65
5.1.2	The Experimental Setup	66
5.1.3	Detecting High-Level Concurrency Errors	69
5.1.4	Fixing the Bugs	71
5.2	Experiments on Multiset	74
5.2.1	Multiset and the Experimental Setup	75

5.2.2	Experiments	75
5.2.3	Test Results	78
5.2.4	Interpretation of Test Results	79
5.2.5	The Effect of Random Thread Scheduling	79
5.2.6	How to Use the LP Coverage More Effectively	81
5.3	Experiments on moldyn	81
5.3.1	moldyn and Experimental Setup	81
5.3.2	Building the SP_{imp} set	83
5.3.3	Experiments	83
5.4	Experiments on SOR	87
5.5	Comparing the Line and the LP Coverage of Benchmarks	88
5.6	Significance of the Experiments	89
Chapter 6:	Conclusions & Future Work	91
6.1	Summary	91
6.2	Future Work	92
	Bibliography	95

LIST OF TABLES

LIST OF FIGURES

2.1	Protecting the critical section using synchronized	9
2.2	Synchronizing the entire method	10
2.3	Using Locks in Java	10
2.4	The producer-consumer example	12
2.5	The Data Race Example	15
2.6	The Interleaving that Leads to a Data Race	16
2.7	Bank Account Example	18
2.8	An execution trace that leads to an atomicity violation	19
2.9	StringBuffer append Method	22
2.10	The execution trace that leads to an atomicity violation	22
2.11	The Happens-before relation	30
3.1	The execution trace that hides the concurrency bug but accomplishes 100 % line coverage	41
4.1	Locations in a Java program	48
4.2	Execution trace of a Java program	49
4.3	An Uncoverable Pair	52
4.4	Constructing Coverable Pairs Set	57
4.5	The relation of sets	59
4.6	Measuring Coverage	62
4.7	The measurement algorithm	63
4.8	Reporting Coverage	64
5.1	The run method of the FTPServer	67

5.2	The close method of the FTPServer	68
5.3	Creating two connections	70
5.4	The proposed fix for the bug of the FTPServer	72
5.5	<code>delete()</code> method of multiset	76
5.6	<code>LookUp()</code> method of multiset	77
5.7	<code>InsertPair()</code> method of multiset	78
5.8	Line and LP coverage with 16 threads in Normal Scheduling and Random Scheduling modes on multiset	80
5.9	LP Coverage with different number of threads on multiset	82
5.10	An uncoverable pair from moldyn	84
5.11	LP and line coverage with 4 and 8 threads on moldyn	85
5.12	A code sample from moldyn	86
5.13	Assigning computations at SOR	88
6.1	A sample code to show the importance of context sensitive analysis	93

NOMENCLATURE

LP	Location Pairs
JVM	Java Virtual Machine
JPf	Java PathFinder
CFA	Control Flow Automata
BLAST	The Berkeley Lazy Abstraction software verification Tool
Promela	Process Meta Language
LTL	Linear temporal logic
SLIC	Specification Language for Interface Checking
VYRD	VerifYing concurrent programs by Runtime Refinement-violation Detection
POR	Partial Order Reduction
ESC/JAVA	Extended Static Checker for Java
JML	Java Modeling Language
DU-Paths	All definition-use path
CP	Coverable Pairs Set
SP	Statically Detected Pairs Set
SP_{imp}	Improved Statically Detected Pairs Set

Chapter 1

INTRODUCTION

1.1 Goals and Motivation

As concurrent software is getting more common, errors due to concurrency are becoming a central issue in software development. Concurrent systems have components that share some resources and can operate on these resources at the same time. Accessing a shared resource may introduce some problems to these systems. Concurrency errors can be very difficult to detect and may have serious consequences such as null pointer dereferences, data corruptions or system crashes.

Verification and testing of concurrent systems may be difficult because the non-deterministic interleavings of concurrently running components may create a huge state space. Race detection, exhaustive testing and model checking tools use techniques proved to be useful for checking the correctness of concurrent software. However, all of these methods have some drawbacks. Model checking and exhaustive testing tools suffer from the state space explosion problem because they try to explore *all* interleavings that may be encountered during the lifetime of a program. Traversing the entire state space may not be able to terminate when the program size is large and therefore, they are not scalable for industrial-scale software.

Even though formal methods are gaining importance in verification of concurrent systems, testing is still a widely used and scalable alternative to check the correctness of concurrent programs. There are some challenges in testing concurrently running components:

1. The state space is huge due to the interleavings of concurrently running components. Different than the sequential programs, the output of the concurrent programs is also dependent on the scheduling of threads. There is a non-determinism in the scheduling of the execution of the threads and executing a particular concurrent component at a scheduling point may be the source of a concurrency bug. As a result, each interleaving that may be encountered should be tested to find a potential bug. As the number of concurrently running components increases and the program size gets larger, the state space grows exponentially. This makes the testing of concurrent programs especially challenging.
2. The bug creating scenario may take place following a rarely encountered execution and that interleaving may never be discovered by test programs. In particular, for Java programs, the Java Virtual Machine (JVM) may not create all interleavings during the execution of the tests and it may be impossible to cover much of interleavings by running the tests many times [1]. It is possible to create all interleavings by deploying code. In order to do this, the tests should be monitored and unexplored interleavings should be extracted. The developer should also direct the programs to get the unexplored interleavings. This may also be an infeasible target because of the huge state space of concurrent programs.
3. When a bug is encountered during a test, replaying the test and finding the source of the bug is not an easy task and may require a lot of effort [1]. For this purpose, the execution of tests are recorded and whenever a bug is found, the recorded execution is re-executed. This is not helpful for finding a bug but for detecting a bug when it is found.

These factors make testing of concurrent programs more challenging compared to the testing of sequential programs. Even when a bug is caught, more effort is needed to detect and fix this bug in concurrent programs.

Coverage analysis is one of the key techniques for increasing the quality of testing. A coverage metric is a set of requirements that a test program has to satisfy in order to measure the exhaustiveness of the testing. Many coverage metrics have been proposed in the literature to improve the testing of sequential programs, but there is relatively little work on coverage metrics which target testing of concurrent programs.

In this thesis, we introduce a coverage metric which is designed to work on realistic configurations of industrial-scale concurrent programs. The LP coverage metric is inspired by an execution pattern which causes high-level concurrency errors. High-level concurrency errors are different than low-level data races in granularity of the code chunks that should be protected to avoid them. A high-level concurrency error occurs when a set of thread-shared variables which must be accessed atomically is accessed non-atomically by a thread. Even in the absence of low-level data races, there may exist high-level concurrency errors in the code [2]. High-level concurrency errors may be hard to detect because the tester should know the code blocks that should be made atomic. The pattern that leads to high-level concurrency errors can be described as follows: A thread t_1 executes a particular line ln_1 and a context switch occurs. Then, another thread t_2 takes the control of the execution and executes another particular line of code, ln_2 . That sequence of events and the dependency between the variables accessed by ln_1 and ln_2 potentially creates a concurrency bug. These sequence of events may lead to a bug because the developer may assume that ln_1 is executed in an atomic block and ln_2 is executed in another atomic block but the code is not well-designed to guarantee assumed atomicity. and Even when the accessed variables are protected by proper locks, this error will occur. This pattern of concurrency bugs are found on several industrial-scale examples and in the literature [3, 1, 4]. The Location Pairs Coverage metric aims to detect if this pattern of two consecutive actions is executed. This way, the developer can test unforeseen scenarios that may lead to high-level concurrency errors.

Symptoms of bugs are conditions that are potential evidence of a bug. The symptoms may be neither the conceptual source of the error nor its manifestation. Modi-

fixing code by focusing on removing a symptom may cause other bugs or symptoms. For example a null-pointer dereferencing is a symptom of a bug and it may be caused by a high-level concurrency error. Only focusing on avoiding null-pointer dereferencing may mislead the developer to treat the bug and the existence of the high-level concurrency error may lead to other problems such as data corruption.

Some of the tools such as race detectors are aimed to detect the symptoms of the bugs. As explained above, removing only symptoms of the bugs may not be sufficient to ensure the correctness of the program. Therefore, focusing on low-level data races found by a race detector is usually not enough. The developer usually has higher level of correctness criterion in mind which aims to make certain code chunks or operations atomic. For most of the programs, users does not make it explicit what code blocks should be made atomic. The LP coverage metric and a race detection tool can help the developers to understand the code chunks that should be made atomic. Different than race detectors, the LP coverage metric aims to find high-level concurrency errors. Because it is designed based on a pattern of a real concurrency bug that leads to concurrency errors, exercising all location pairs increases the chances of detecting a high-level concurrency error. Refining the code using the LP coverage reports to test the intended behavior makes the developer to think about the atomicity of the code blocks in detail. In short, using the LP coverage metric makes the testing process interactive and enables the developer to think about the program in more detail and refine her intent.

The LP coverage method does not suffer the problem of exponential blow-up of the state space because it does not require to traverse the state space. The LP coverage metric requires pairwise consideration of accesses to shared variables. Thus, the size of the coverage tasks is quadratic in the size of the program. Unlike model checking, it can practically be used on large-scale software. Using the LP coverage may also help automated techniques and formal methods by providing coverage targets. For example, a test case generator that uses the reports of the metric can execute interesting and rarely observed scenarios and can detect bugs.

There are three phases of the LP coverage metric: constructing the set of coverage tasks, measuring the coverage and reporting the coverage. We construct the set of coverage tasks using a race detection tool Chord [5]. We detect each access to thread-shared variables using static analysis of the Chord and build a coverage set. We measure the coverage by monitoring the execution of the tests dynamically. We implemented a dynamic monitoring tool on a model checker, Java PathFinder (JPF) [6] to measure the LP coverage of testing programs. The tool operates on Java bytecode and runs the Java program on a virtual machine (JPFs JVM). Then, we trace the execution of the program and examine to see whether we encounter any execution trace with the same pattern that leads to high-level concurrency errors as explained above. Finally, the covered and uncovered LP pairs are reported to the designer. The designer may need to refine the set of coverage tasks as she gets coverage reports and modify and repeat the tests accordingly.

The tool also has a random scheduling mode which directs tests to get more interleavings. We run a number of experiments on various-scale Java benchmarks and showed the following facts about the LP coverage metric:

1. Existing testing metrics for sequential programs do not handle concurrency adequately.
2. The LP coverage metric assists the developer to create interesting scenarios to test all possible interleavings that may cause high-level concurrency errors.
3. The metric can be used on large-scale programs.

1.2 Contributions

1. We present a coverage metric for concurrent programs. This work is one of a few coverage metrics in the literature which targets concurrency. Even though there was extensive research on coverage metrics for sequential programs, few methods have been proposed for concurrent systems. Some techniques were

presented but their scalability and usability were never tested [7, 8, 9]. Our technique is implemented and experimented on real concurrent programs.

2. The LP coverage metric is a scalable and practical metric which has a polynomial running time proportional with the number of thread-shared variables. Our technique requires to track accesses to shared variables during the lifetime of a program. We show that this introduces a small burden on the analysis. Since we do not require that all paths or states of the program is required to be visited, and we only monitor the execution, our method can be used practically.
3. Some of the practically used verification tools focus on low-level symptoms of concurrency bugs. For most cases, removing the symptoms of a bug is not enough to ensure the correctness of a concurrent software. Low-level data races are usually symptoms of concurrency bugs in concurrent programs. The designers usually have a higher-level correctness criterion in mind and for ensuring correctness, they should make certain code blocks and operations atomic. Our tool is inspired by an execution pattern that leads to high-level concurrency errors. We show that exercising all LP pairs increases the chances of detecting high-level concurrency errors. The LP coverage metric makes the testing process interactive and enables the developer to think about the program in more detail about the atomicity of the certain code blocks and operations. This way, it helps designers to test the programs according to the intended behavior.
4. We implemented the LP coverage metric and tested it on Java programs. We have shown that the metric can be used to detect concurrency bugs on real Java programs. Also, we have shown that the coverage reports can assist the developer to test rare and unforeseen scenarios on these programs. We have also demonstrated that the technique is a scalable technique for large-scale programs which model checkers cannot handle.

1.3 Organization

The organization of this thesis is as follows: In Chapter 2, we present concurrency concepts and concurrency related bugs along with a survey on techniques used for verification of concurrent software. Chapter 3 presents how coverage is used for improving testing and gives a brief survey about sequential coverage techniques. It explains why sequential techniques are not helpful for testing of concurrent programs and also presents some existing coverage techniques for concurrent programs in Chapter 3. Chapter 4 presents the LP coverage metric algorithm and explains how it is implemented. Experimental results of the metric on various-scale Java benchmarks is presented and evaluated in Chapter 5. Chapter 6 presents the future work and concludes the thesis.

Chapter 2

CONCURRENCY AND CONCURRENCY BUGS

Concurrent programming is becoming more common in software development. Widespread usage of multiprocessor systems and server-side applications has made it inevitable to use concurrent software. Due to the widespread usage, concurrency bugs became a central issue for developers. This chapter introduces what concurrency is and what the causes of errors that developers might have in their concurrent programs are. This chapter also gives a brief survey about some techniques and tools used for the verification of concurrent programs.

2.1 Correctness for Concurrent Software

2.1.1 General Concurrency Concepts

Concurrency in computer science is the running of two or more processes at the same time on a single or multiprocessor system. A common example of concurrent systems is a web server. A web server can serve more than one client at the same time by creating concurrent connections. A concurrent system may contain multiple threads or processes. The basic difference between a thread and a process is that processes have separate address spaces whereas threads use a shared address space. Usually threads are defined as light processes, because creating threads are less costly than creating processes. Throughout this thesis, we will focus on multithreaded systems but most of the concepts are valid for multiprocessor systems too. There are two communication methods between processes of a concurrent system: message passing and shared memory. In message passing communication, processes have separate memory spaces and send information to each other via channels. In shared memory systems, processes or threads share a portion of the memory and they all can change

```
Object lock = new Object();  
synchronized(lock){  
    critical section ...  
}
```

Figure 2.1: Protecting the critical section using `synchronized`

the content of this memory by operating on *shared variables*.

2.1.2 Concurrency in Java

In this thesis, we present a coverage metric and an algorithm for it which can be applied on shared memory multithreaded systems. We implemented and tested our software on Java. In this section, basic concurrency constructs for Java are introduced. Even though some of these constructs are specifically designed for Java, most of the concepts are also used in widely used programming languages such as C or C++. Detailed information about Java concurrency constructs can be found in [10, 11].

In concurrent programs, threads operate on shared variables and the order of accesses to these variables may affect the overall behavior of these programs. A *critical section* is a code block where shared data should not be accessed by multiple threads. If concurrent accesses of multiple threads to critical sections are not prevented, the shared data may be corrupted or may go to an unstable state. In multithreaded programs, in order to serialize accesses to resources (such as memory or a database), mutual exclusion must be ensured. When mutual exclusion is ensured in a multithreaded program, concurrent accesses to critical sections are avoided. The mechanism to ensure mutual exclusion in Java is *monitors* and *lock based monitors* are used for protecting code blocks. Java provides different locking mechanisms to the developer. First, the developer can create a lock object and can protect the critical section by locking this object using the `synchronized` keyword as shown in Figure 2.1.

Another mechanism that Java provides via synchronization construct is to syn-

```
public synchronized void method(args){  
    critical section ...  
}
```

Figure 2.2: Synchronizing the entire method

```
private final ReentrantLock lock = new ReentrantLock();  
public void method() {  
    lock.lock();  
    try {  
        critical section ...  
    } finally {  
        lock.unlock()  
    }  
}
```

Figure 2.3: Using Locks in Java

chronize an entire method. In Java, each object has a lock associated with it. If a method is declared to be synchronized, the lock of the object whose method is called is acquired. Thus, two threads cannot access two synchronized methods of an object concurrently. A method can be synchronized as shown in Figure 2.2.

Java provides other kinds of locking constructs with Java 5.0 under `java.util.concurrent.locks` package. This package offers constructs used for locking and synchronization purposes other than built-in constructs such as the `synchronized` keyword. One can create different types of locks such as `ReentrantLock` or `ReadWriteLock` to ensure different kinds of protection. The usage of these constructs are as shown in Figure 2.3.

When a thread holds the lock of an object, no other thread can get the same lock. This is how mutual exclusion is guaranteed in Java. A thread is said to be *blocking*

if it holds the lock of an object and prevents other threads to get the same lock and to enter a critical section protected by this lock. A thread is called to be *blocked* if it cannot enter a critical section because another thread has already acquired the lock of this critical section.

Java provides conditional constructs which allow thread synchronization at higher levels in addition to mutual exclusion. A thread that holds the lock of an object may call *wait()* method of that object and releases the lock of that object to enable other threads to acquire this lock. That threads goes in an idle state until some condition holds and another thread awakens it by calling *notify()* method of the same locked object. A famous example of thread synchronization using monitors is producer-consumer problem shown in Figure 2.4. In this problem, there are two or more threads concurrently running. The producer threads inserts some elements into the shared buffer by calling the *put()* method, and some concurrently running consumer threads take some element out of this buffer by calling the *get()* method. If the shared buffer is empty, there are no elements to consume. The consumer thread will be suspended and waits until the producer puts an element in the buffer and calls *notify()* to wake it. On the other hand, if the shared buffer is full, the producer thread will be suspended and wait until the consumer removes an element from the buffer and calls *notify()* to wake it. When the buffer becomes available and the *notify()* method is called, the thread scheduler of the virtual machine picks a random thread among suspended threads and executes it. *notifyAll()* method of an object wakes up all the threads that are suspended on that object.

Thread is a special Java class which provides several methods for controlling the Java threads. Calling **Thread**'s static method *sleep(millisecs)* causes the current thread to wait for specified number of milliseconds. Another static method of **Thread** class, *yield()* is used to pause the execution of the current thread and forces the virtual machine to pick an available thread. Note that the scheduler may pick the same thread that has yielded, so it does not guarantee a thread context switch. The threads in Java also have priorities. When multiple threads are available at a

```
public class SharedBuffer{

    Element[] buffer = new Element[10];
    boolean available = false;

    public synchronized int get(Element element) {
        while (buffer.length == 0) {
            wait();
        }

        removeFromBuffer(element);
        notifyAll();
    }

    public synchronized void put(Element element) {
        while (buffer.length == 10) {
            wait();
        }

        addToBuffer(element);
        notifyAll();
    }
}
```

Figure 2.4: The producer-consumer example

scheduling point, the virtual machine picks the thread with the maximum priority and runs it. The priorities of the threads can be changed during the program using the *setPriority(priority)* method.

Java also provides higher level of concurrency constructs such as *executors* and *thread pools* and other constructs to manage them with Java 6.0 but they are not dealt with in this work.

2.1.3 Concurrency Problems

The correctness properties of concurrent systems can be classified into two main categories: safety and liveness properties. These concepts were first introduced by Lamport in [12]. Safety properties state that something bad will not occur during the execution of the program and liveness states that some good thing will happen during the execution.

- **Safety Properties:** As an example of safety property in concurrent systems, *deadlock freedom* requires that the concurrent system does not go into deadlock state. Deadlock occurs when a thread is blocked and cannot proceed because it also blocks the thread which is expected to wake it. Since no one can make progress, they wait forever and the system stops functioning. Livelock is similar to deadlock but in livelock even though threads are active, none of them can proceed because they continuously change their states. Another safety property for concurrent programs is *race freedom*. A data race occurs when a variable is accessed by more than one thread, where at least one writes to it and there is no happens-before relationship between the accesses [13]. If a data race occurs during the execution of a program, it may cause a null pointer dereference or data corruption. As a basic correctness criterion for concurrent Java programs, race-freedom ensures a total order on operations on data variables. Data races are among the most common problems encountered in concurrent systems.
- **Liveness Properties:** *Fairness* is a liveness property for concurrent systems.

Fairness guarantees that the next operation of each thread will be executed eventually in a system. If the system does not guarantee fairness, one or more processes may suffer from *starvation*. Starvation happens when one or more threads of the system cannot make progress because threads' operations are not executed.

The focus of our coverage metric is on races caused by interleaved operation of threads on shared variables rather than starvation or deadlock. In the next section, various types of data races will be introduced in detail and the rest of this thesis will target programs that may have this kind of concurrency bugs.

2.2 Concurrency Bugs

2.2.1 Low Level Data Races

If a race condition exists in the program, the output of the program is dependent on the interleaving of threads which access it. Some data races may be overlooked by the developer easily and they can be very harmful if an unforeseen interleaving takes place. Simple data races occur in the program because usually the developers assume that single line of codes are operated atomically. Let us explain what *atomicity* means and how races occur in a simple example.

In the example shown in Figure 2.5, `increment()` and `decrement()` methods of the class are used for very basic tasks, to increment and decrement the value of a private variable of this class. `x++` operation is not an atomic operation. In the bytecode, there are several instructions to increment the value of `x`: loading the variable in the operand stack from memory, incrementing its value and writing the value back to memory. Decrement operation has similar steps too. Suppose that two threads attempt to access to the same instance of `MyClass` and to increment and decrement the value of variable `x`. If the interleaving shown in Figure 2.6 takes place, an inconsistent value will be present in the memory. The initial value of `x=0`, and increment and decrement operations are called by two threads. The user expects the

```
public class MyClass {  
    private int x;  
    ...  
    public void increment( ){  
        x++;  
    }  
    public void decrement( ){  
        x--;  
    }  
}
```

Figure 2.5: The Data Race Example

two methods to be run atomically and whatever the interleavings are, she expects that the value of x will be incremented and decremented serially. Therefore, the value of the variable x after these two operations are expected to be equal to its initial value. As seen in Figure 2.6, the value of the variable in the memory is set to -1 at the end of the execution.

This kind of data races are called low-level data races in [2] because they occur because of interleaved instructions on a single variable. Traditionally, low-level data races are avoided by using locking mechanisms. In Java, as explained in Section 2.1, explicit locks or the **synchronized** keyword can be used in order to avoid races. In the example presented above, locking the same lock object for **x++** and **x--** operations or synchronizing the **increment()** and **decrement()** methods will prevent the race condition. So, data races on shared variables can be avoided by serializing all accesses to these variables by acquiring the lock before accessing these variables and releasing after accessing them [2]. Data races may have unexpected effects on systems and may cause critical system errors such as null pointer dereferencing, data corruption or even system crashes.

initial value of x is 0

Thread 1	Thread 2
-----	-----
call increment()	call decrement()
load x	--
--	load x
x = 1	--
write to memory (x=1)	--
--	x = -1
--	write to memory (x=-1)

the value of x after operation is -1

Figure 2.6: The Interleaving that Leads to a Data Race

Even though all shared variables of a program are lock protected and the program is free of data races, the program may still contain some concurrency errors. In the next subsection, a higher level correctness requirement that the developer might require as a correctness criterion in mind will be introduced.

2.2.2 High Level Data Races

As described in [2], a high-level data race occurs if a number of shared variables must be updated atomically but a thread may change some of them non-atomically. That definition implies that focusing on individual accesses to single variables may not be adequate in order to avoid concurrency bugs in concurrent software. The developer has a higher level correctness criterion in mind that may require to update a set of variables, and a sequence of statements of the program atomically.

- **Atomicity:** An atomic code block refers to a block where the execution of that block cannot interfere with concurrently executing threads [14]. As explained in [14], atomicity is a widely used criterion for the correctness of concurrent programs and using Java's `synchronized` keyword is a commonly used design practice to make entire code blocks atomic. Race-freedom and atomicity can be provided using proper locks but the difference between them is the granularity of protected blocks. In order to avoid low-level data races, only individual accesses to shared variables should be protected whereas in order to ensure atomicity, code blocks should be accessed atomically. Atomicity violations may cause severe problems like null pointer dereferencing, data corruption or system crashes.

Even though a program is race free, it may still contain atomicity violations. Let us illustrate this point on a *Bank Account* example which was presented in [15]. The example is shown in Figure 2.7.

The example presented in Figure 2.7 is a bank account which is accessed by multiple threads and the variable `balance` is the shared variable. The `update()`

```
1 class Account {
2     int balance;
3     synchronized int read() {
4         return balance; }
5     }
6     int update(int a) {
7         int tmp = read();
8         synchronized(this){
9             balance = tmp + a;
10        }
11    }
12}
```

Figure 2.7: Bank Account Example

method can be run concurrently by multiple threads and is used to update the balance by adding the method's argument to current value of the **balance**. As seen in this example, there are two accesses to the shared variable **balance** at line 4 and at line 9 of the code. Both accesses are protected by the same lock and there is no data race on **balance**. Consider that the execution shown in Figure 2.8 took place. The initial value of **balance** is 100 and two update operations with 200 and 300 are called by two different threads. Since update operation adds the argument value to the current value of **balance**, **balance** should be set to 600 after the operations. However, when these update operations are completed following the execution trace presented in Figure 2.8, the final value of the variable is 400. This is a typical atomicity violation and caused by non-atomic update of the shared variables performed by the improperly synchronized code block.

Atomicity is an appropriate correctness criterion for many concurrent programs.

initial value of balance is 100

Thread 1	Thread 2
-----	-----
call update(200)	call update(300)
tmp = 100	--
--	tmp = 100
balance = 100+200 = 300	--
--	balance = 100+300 = 400

the value of balance after operations is 400

Figure 2.8: An execution trace that leads to an atomicity violation

Still, ensuring race-freedom and atomicity of the concurrent program may not be sufficient for correctness of concurrent programs. In particular, making large code blocks atomic may lead to an unacceptable performance hit and can even prevent the program from functioning correctly [4].

- **Refinement:** If the correctness requirement for a concurrent program is atomicity, the sequential specification is based on the implementation [3]. Atomicity requires that an atomic code block protected by a particular lock is started and finished without any interference with other atomic blocks protected by the same lock. Consider an example of a method that can be terminated exceptionally because of the resource contention of the running threads [16]. If the correctness of this implementation is considered based on atomicity, there should be no resource contention between threads because only one thread can enter the atomic methods at a time. Thus, the specification which is based on atomicity does not reflect the specification of the real implementation. When testing the system, this expected behavior of the system will be reported as a

software bug according to atomicity criterion. Because in the case of atomicity, the specification is not separate from the implementation, it may be a very restrictive criterion for concurrent programs.

A more permissive correctness criterion, *refinement* requires that only an abstract view of the program and not the entire state itself be updated atomically. *I/O refinement* is defined in [17] as follows: refinement is the requirement that the return values of the method calls in the atomic execution of the program should be consistent with the return values of the method calls of the atomic execution of its specification. The ability to use specifications that models the behavior of the system more precisely makes refinement a more appropriate correctness criterion for concurrent programs [3]. In [3], an algorithm was introduced for checking refinement violations. For checking the I/O refinement, the user annotates the code by inserting commit points where the methods return and commit their actions. So, for the I/O refinement checking algorithm, the specification is based on the intended commit points of the concurrent program. Another definition for refinement, the *view refinement* is also introduced in [16] as follows: Not only the return values at commit points, but also the programmer-defined abstract states of the implementation and the specification should be consistent at commit points. Since the view refinement definition requires the consistency in entire state of the program, it provides more rigorous checking than I/O refinement.

As explained in this section, low-level data races are not the only cause of concurrency errors due to unexpected interleavings of processes. The target of the LP coverage metric, introduced in this thesis, is to capture high-level concurrency bugs such as atomicity and refinement violations.

2.3 Motivating Example

In this section, we introduce a real example which has a high-level concurrency error. Later, we will refer to this example to show how the LP coverage metric can be used for detecting concurrency errors. This example is an older version of a Java library, `StringBuffer`. Figure 2.9 presents the `append` method of the older version of `StringBuffer`. The `append(arg)` method adds the `arg` to the string. At this implementation, the `append` method has a concurrency error because even though the method itself is synchronized, its argument is not locked throughout the method [14]. At line 1 in Figure 2.9 the length of `sb` is saved to the variable `len`. At line 2, `newcount` is calculated by adding `len` to `count` which keeps the length of `this`. At line 3, `value` is the `this`'s current value before appending `sb` to it. At line 3 and 4, it is checked if new `StringBuffer`'s length will be bigger than the current one, and it is expanded accordingly. At line 5, `sb.getChars(int srcBegin, int srcEnd, char[] dst, int dstBegin)` copies `sb`'s characters are appended to array `dst` starting from index `dstBegin`. The characters of `sb` are copied between indexes `srcBegin` and `srcEnd`. At line 6, the new length is saved and at line 7, the new `StringBuffer` is returned.

Let us explain how this error can occur: during the execution of an `append` method by a thread, say t_1 , the content of `sb` can be modified by another thread, say t_2 as shown in Figure 2.10. In that execution trace, after t_1 invokes `sb.length()` at line 1 of the `append` method and it saves the length value to local variable `len`, t_2 may call `sb`'s `setLength(int newLength)` method of `sb` with `newLength` value 0. When t_1 executes and invokes `sb.getChars(0, len, value, count)`, an `ArrayOutOfBoundsException` exception will be thrown because the original value of `len` was modified. Notice that this error occurs whenever line 1 of `append` method followed by an instruction that modifies the length of `sb` before line 5 of `append` method is executed.

The error above is particularly caused by an atomicity violation. This pattern of two particular consecutive execution of two dependent actions from two method bodies explains all atomicity and refinement violations [18]. The LP metric is inspired

```

public synchronized StringBuffer
    append(StringBuffer sb) {

1      int len = sb.length();
2      int newcount = count + len;
3      if (newcount > value.length)
4          expandCapacity(newCount);
5      sb.getChars(0, len, value, count);
6      count = newcount;
7      return this;
8  }

```

Figure 2.9: StringBuffer append Method

t1	t2
-----	-----
invoke sb.length();	--
int len = sb.length();	--
--	invoke sb.setLength();
--	sb.setLength(0);
int newcount = count + len;	--
if(newcount > value.length)	--
invoke sb.getChars();	--
sb.getChars(0, len, value, count)	--

Figure 2.10: The execution trace that leads to an atomicity violation

by this pattern and specifically targets to capture similar errors in Java programs.

2.4 Verification Techniques

In this section, we will introduce some techniques for checking correctness of concurrent software which are relevant to our work and we will briefly explain their strengths and weaknesses.

2.4.1 Static Techniques

Static techniques examine the concurrent program and extract some useful information about the software for verification or bug hunting purposes.

- **Model Checking:** Model checking is one of the most commonly used software verification technique. Model checking traverses the state space of a concurrent system and explores concurrent executions for errors. It has three phases: modeling, specification and verification. The first task is to construct the model of the system. In the specification phase, the properties that will be checked is formulated as predicates over all executions or in temporal logic. In the verification phase, the checker traverses the state space constructed over the model and checks if properties hold [19]. At the end of the check, the model checker presents execution traces called counterexamples to the user. The main advantage of model checkers is that they explore all states visited during the execution in an automated way. However, what makes the model checking approach superior becomes a bottleneck for most of the practical cases; since concurrent systems may interact in many different ways, traversing the entire state space of the program is an expensive task. Usually, model checking a real system does not terminate in a reasonable amount of time and it requires huge memory resources. Thus, challenging the state space explosion problem is one of the main issues of model checking [19].

There have been various model checkers in the literature with various capa-

bilities. BLAST [20] checks for C programs which are represented as control flow automata (CFA). It constructs a reachability tree using the CFA of the program and tries to reach error labels on that reachability tree. BLAST uses lazy abstraction [21] on-the-fly in order to refine the CFA. BANDERA [22] takes a Java program and converts the program to a model in the *Promela* language, a special modeling language for SPIN model checker [23]. BANDERA also enables the user to provide a temporal specification language to annotate their properties. These specifications are also translated to model-checker's language. In both BLAST and BANDERA, the tools expect the user to guide the abstraction of the code. SPIN is a model checker based on Büchi automata. In SPIN, the properties are provided using LTL formulas, but some basic liveness and safety properties can be checked on-the-fly without requiring LTL formulas. SLAM [24] takes a C program and instruments it. It then uses the instrumented program in order to build a boolean program using predicate abstraction. At the next step, it uses a technique called *courterexample-driven refinement* for building a more precise boolean program. It tries to reach error labels same as BLAST on that refined boolean program and if an error label is reached, it checks if it is feasible or not. SLAM does not require user annotations, because safety properties are already encoded in a language called SLIC (Specification Language for Interface Checking) [24]. SLAM is used for verifying safety and liveness properties of sequential programs.

Even though we classified model checkers as static analyzers, there are some model checkers that take the program and check the properties by running it. So, they can be classified as dynamic analyzers. Java Pathfinder (JPF) [25] is a Java model checker that we also used in order to implement our coverage metric. JPF has its own virtual machine which take the Java program as input and runs it as is without any abstractions. JPF checks for deadlocks and user specified properties via annotations and LTL formulas. To avoid visiting a state more than once, JPF saves each visited state hashed and checks whether a state was

visited before. When an erroneous state is encountered, JPF produces a trace report to the user. JPF also uses partial order reduction (POR) to tackle the state space explosion problem. Verisoft [26] is another tool that can be classified as a dynamic checker. It is a stateless model checking tool that uses POR on-the-fly for reducing and traversing the state space. Verisoft checks for deadlocks and user specified assertion violations as well as divergences and livelocks in C and C++ programs.

Along with predicate abstraction, partial order reduction (POR) is one of the most successful techniques to handle state space explosion problem in model checkers. The basic idea behind this technique is to not visit the states which are equivalent to each other. Instead of constructing and traversing the entire state graph of the program, a selected subset is constructed and traversed which is sufficient to prove the correctness of the program [19]. This technique is called selective search and it is the approach behind many partial order reduction algorithms [26]. The partial order reduction techniques are based on the concept of dependency. Two transitions are dependent if they access the same shared variable where at least one writes to that variable. Model checking algorithms have different implementations of POR. The persistent/stubborn sets of Godefroid [27], the sleep sets technique described in [27] and ample sets [19] are among the most well-known techniques (usually combined together) for reducing the state space traversed.

Another idea to tackle the state space explosion problem is using parallel or distributed programming. Two examples of parallelized model checkers are SPIN model checker and Mur ϕ Verifier [28]. Both tools apply a similar approach with JPF in order to reduce state space and traverse it. They save the visited states in a hash table and whenever they counter a new state they check whether it has been visited before. Roughly speaking, the parallel state space search algorithm partitions the state space among the available processors and each processor makes a state search on its own state domain. Each domain should

have knowledge about other domains to prevent visiting an already visited state. This is accomplished by using messages about visited states between processes.

- **Static Analysis:** Static analysis observes the system behavior without executing the program actually. Pointer analysis is a commonly used static technique for especially sequential programs but it has also been used for multithreaded programs recently. Pointer analysis determines at compile time which locations may the pointers point to during the execution of the program [29]. Exact pointer analysis is undecidable and there are some trade-offs between various types of pointer analysis. Some methods are precise but expensive whereas some other methods are less expensive but produce less precise results. Flow-sensitivity [30], context-sensitivity and heap-modeling [29] are some of the constraints that determine the precision of pointer analysis. Roughly speaking, if a pointer analysis is flow-sensitive, it uses the control flow information of the program and follows the order that the statements are executed. In a context-sensitive analysis, the analysis considers the calling context of methods (or functions). The main difficulty of applying current static analysis techniques to multithreaded programs is that, in multithreaded programs, concurrently running processes may interleave in many ways and this increases the chance of a pointer to point a memory location. That may affect the precision of the analysis dramatically. Rugina and Rinard developed a flow-sensitive and context-sensitive pointer analysis for multithreaded programs and presented their algorithm in [31]. In their algorithm, they use *interference information* which captures the effects of pointer assignment by parallel threads and they prune infeasible pointer aliases using this information.

One of the most common usages of pointer analysis and other static analysis of multithreaded programs is for the detection of data races. Two of the popular static race checkers are Warlock [32] which targets C programs and ESC/JAVA (Extended Static Checker for Java) [33] which targets Java pro-

grams. ESC/JAVA is a popular static checking tool which can be used for several purposes other than race detection such as detecting null pointer dereferencing, array bounds exceptions, type cast errors and deadlocks. The user annotates the Java program for desired properties using Java Modeling Language (JML) [34]. The user can annotate which locks guards which variables and code blocks in order to detect races with ESC/JAVA. Warlock also requires the code to be annotated to be checked. RacerX [35] is a static race checker which does not require annotations, and can handle large programs. It uses flow-sensitive, context-sensitive analysis and it also can be used to find potential deadlocks.

The tool that we also use for our static analyzing phase is Chord by Naik, Aiken and Whalkey [5]. Chord is a static race detection tool for Java programs which uses context-sensitive, flow-insensitive analysis. It has four stages for detecting potential races.

1. In the first phase, the tool detects accesses to all fields and arrays. This information is used to detect races because two accesses can be in a data race only if they access to the same variable and at least one of them writes to this variable. It then prunes this information to approximate the set of potential races. The pruning is called the *reachable-pairs computation* and it decides if the accesses to shared variables are reachable from the thread-spawning call sites.
2. The second stage of the algorithm is *aliasing-pairs computation* which decides if two accesses are to the same memory location. Alias analysis computes if two pointers may point to the same location.
3. The third stage, *Escaping-pairs computation* computes if the data accessed is thread-shared data because two accesses may be involved in a race if they are on a thread-shared data.
4. The fourth phase, *unlocked-pairs computation*, determines if two accesses

hold a common lock because two accesses can involve in a race if they are not protected by a common lock.

Choi et. al. uses a similar technique in [36] for detecting data races statically but their analysis is context-insensitive.

The main problem of static analysis is that static tools produce false alarms. Some tools require the user to annotate the code according to the desired functionality but this introduces a considerable burden of annotation to the user. Static tools' main strength comes from their scalability.

2.4.2 Dynamic Techniques

Dynamic techniques monitor the program by executing it on a real system or on a virtual system. Most dynamic tools require instrumentation of the code.

- **Race Detection Tools:** Although we introduced some race detection tools under the static techniques category, most of the state-of-art race detection tools make use of dynamic analysis. *Eraser* [37] is the most popular dynamic race detection algorithm introduced by Savage et. al.. Eraser uses the *lockset* algorithm. The lockset algorithm initializes a set for each variable in the program. A variable's set keeps the locks which protects accesses to this variable. The algorithm updates this set after each access to a shared variable. For example, if a thread, holding only the lock l , accesses to variable v , the lockset is updated by intersecting the locks present in the set and l . If the set ever becomes empty, this indicates that there are some accesses to this variable which are not protected properly. The lockset algorithm is made more precise by recalling some commonly used concepts of multithreaded programming. For example in most of the programs, no lock is acquired to protect a variable during creation. Also, some variables may never be modified during the execution of a program and because they are only read, they do not need to be protected by common locks. Finally, some programs may use read-write locks which may allow threads to

read from the variable concurrently if there is no write operation on that variable. The main problem of the lockset based approaches is the large number of false alarms that the algorithm reports. A lot of tools and algorithms have been proposed based on the lockset algorithm in the literature most of which improve the approach and produce less number of false alarms and report more precise results [38, 39].

Another dynamic race detection technique is based on the *happens-before* relation [40] which uses *vector clocks*. The happens-before relation is the partial order on all events on shared objects. The events are already ordered in a sequential program, or on a single thread. In a concurrent execution, the events between parallel processes are ordered according to the accessed synchronization objects. For example for the execution shown in Figure 2.11, let's say *lock* is a lock object and *v* is a shared variable, and two threads access the variable *v*. In this example, the accesses to *v* can be ordered according to happens-before relation, because both accesses are protected by a common lock and it is impossible for two accesses to interleave with each other [37]. The race detection algorithms based on happens-before relation check if two accesses to a shared variable are not ordered by the happens-before relation [41]. The happens-before approach is a more precise approach compared to lockset approach but it has still some drawbacks. First, it is not efficient because it has to keep a vector of variables of size the number of threads for each synchronized variable. Second, its precision is highly dependent on the scheduling of the threads [37]. A number of tools have been developed which aimed to overcome these drawbacks for detecting races dynamically based on happens-before relation [42, 43, 44].

Some techniques use both lockset and happens-before relation techniques for detecting races [45] [46]. Elmas, Qadeer and Tasiran introduced a new technique, Goldilocks, for detecting race conditions dynamically in [41]. Goldilocks captures happens-before relation using lockset based approach. This is a sound algorithm which does not report false alarms and it is as precise as happens-

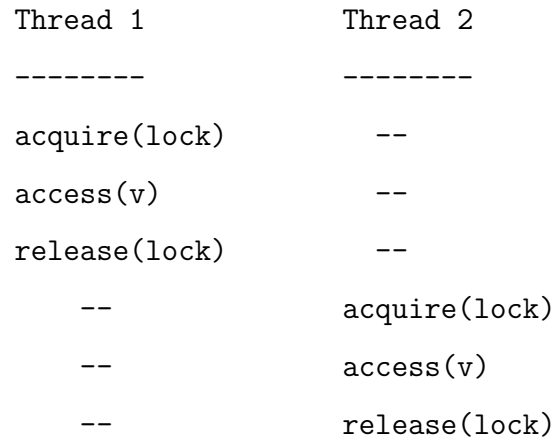


Figure 2.11: The Happens-before relation

before based algorithms and as efficient as lockset algorithm. In order to ensure scalability and efficiency, Goldilock uses lazy computation of locksets and uses short circuit checks which check for the existence of an happens-before edge in constant time. Using these optimizations, Goldilocks obviates the need to keep a separate lockset for each variable and locksets are created only temporarily when a variable is accessed. The algorithm computes the happens-before relation only if the constant time short circuit checks fail. These optimizations make the algorithm a not only precise but also an efficient algorithm for detecting data races.

- **Noise Makers:** Noise makers are tools which create more interleavings during the test of a multithreaded program and they strive to fail the tests using some heuristics. Thus, noise makers are not techniques which find the bugs but they help test cases to be improved [47]. In [9], Edelstein et. al. proposes a technique based on Java synchronization constructs. In their technique, they force the thread scheduler to reschedule threads at points where shared variables and concurrency constructs are accessed. This is achieved by inserting `sleep()` and

`yield()` and by changing the priorities of threads randomly. We use a similar approach in our tests while measuring our metric for producing more interleavings. We achieve this by not inserting thread yields at points where shared fields are accessed but we directly force the scheduler to pick a random thread at virtual machine level. Whenever an instruction which accesses the shared memory is accessed, we pick a random thread among available threads at this point and we execute the selected thread. Stoller implements a similar technique for Java programs in [48] by transforming instrumented Java programs. The transformed program reschedules threads at points where memory are accessed. [49] uses dataracebased heuristics for controlling of threads by using `yield()` statements and barrier points.

- **Replay:** The output of the concurrent programs is dependent on the scheduling of threads. There is a nondeterminism in the scheduling of the execution of the threads because executing a particular concurrent component at a scheduling point may change the output of the program. The threads may interleave in a huge number of ways. A thread interleaving in a test that creates a concurrency error may never be seen again by repeating the same tests more. This becomes a challenge for detecting bugs in concurrent software. In order to overcome this problem, a technique called *replay* is developed. The replay technique consists of two phases; recording and replaying. In the recording phase, the replay tool records the execution of processes of the parallel system and the replay phase runs the execution of the parallel program in a deterministic fashion to debug the code [47]. One of the earliest tools for replaying Java programs is *DejaVu* [50]. *DejaVu* is implemented by modifying Sun's Java virtual machine and it records *critical events* that happens during the execution. Critical events are accesses to shared variables and other synchronization events such as wait and notify. Then it replays the same multithreaded execution using the events recorded.
- **Runtime Monitoring:** Java PathExplorer [51] is a monitoring tool that pro-

vides two functionalities to the user. Java PathExplorer's *logic based monitoring* checks formal specification of the execution according to some high-level logic. It checks if the execution trace of the program conforms to the user supplied formula (such as temporal logic formula). The tool's *error pattern analysis* uses error detection algorithms, such as race detection or deadlock and more algorithms can be inserted in the tool to check for other error patterns. The tool works by instrumenting bytecode according to user-supplied formula or the error pattern algorithm.

Java MultiPathExplorer [52] also targets to detect specification violations using an external observer. The tool instruments the system before its execution and it extracts the relevant events at runtime. The observer analyzes the system by building an abstract behavior of the system using the received messages. It also uses the partial-order of the concurrency related events during the analysis. Using this off-line analysis, not only does it capture the errors of a particular execution trace, but it also can predict potential errors in the system.

There are several runtime monitoring tools which are developed to capture high-level concurrency errors introduced in Section 2.2. Atomizer [14] is a tool that uses Lipton's theory of reduction and lockset algorithm in order to capture atomicity violations. The target program analyzed may be annotated by *atomic* keyword and the atomicity of particular code blocks can be verified using these annotations. Refinement violations can be checked by runtime monitoring using VYRD [3]. VYRD can check for two refinement criteria. For I/O refinement, it checks the return values of methods in the implementation and in the specification at commit points. For view refinement, it checks the conformance of the program state to the state of the specification based on the commit points of methods.

We explained techniques which deploy systematic state space exploration under the model checking category. Tools like JPF and Verisoft visit all states that

can be encountered during the execution of a multithreaded program and they explore user-annotated properties or error patterns like races or deadlocks. In order to avoid the state space explosion problem, some techniques such as partial order reduction are proposed for these systems.

2.5 Focusing On Developer's Intent

We introduced the low-level and high-level data races in Section 2.2. In most cases, absence of low-level data races does not ensure the correctness of the multithreaded programs and higher level atomicity should be considered. The usual solutions to avoiding concurrency errors in Java programs involve using synchronization mechanisms such as monitor-based locking. For large-scale applications, avoiding concurrency errors and achieving high performance are conflicting elements of software development. Protecting large pieces of code using locks often has an unacceptable impact on performance. On the other hand, protecting too small a block of code or too few objects with locks often leads to subtle concurrency errors.

For these reasons, making precise the developer's intent and identifying the corresponding correctness criterion is of key importance. The most appropriate fix for a concurrency error can only be determined based on this information. Further, the program analysis tool that is best suited for a system is one that takes the appropriate correctness criterion as its basis. As explained in Section 2.4, most of the tools focus on low-level races of multithreaded programs. On the other hand, verification tools which target to find high-level errors sacrifice efficiency or they depend on user annotations.

The concurrency errors can be identified by the following aspects:

Bug manifestation: This is the observed discrepancy in program behavior caused by the bug. Examples are null pointer dereferences and data corruption. While some of these errors can be detected upon their occurrence by the runtime system, others, such as data corruption, may go undetected. If a tool targets to find the manifestation of the bug, removing this manifestation may not be helpful for removing the real bug.

For example to avoid a null pointer dereferencing at some point of the code caused by data race does not heal the race and the race may cause other violations such as data corruption.

Symptoms detected by tools: These are conditions that are potential evidence of a bug. Program analysis tools that target concurrency errors have as their focus a category of symptoms, such as race conditions or code patterns frequently associated with concurrency bugs. The association between bugs and these symptoms is often indirect. The symptoms may be neither the conceptual source of the error nor its manifestation. Frequently, modifying code by focusing on removing a symptom may introduce other bugs and/or symptoms. We gave an example of a situation where only removing data races does not guarantee the atomicity of the required code blocks in Section 2.2. Focusing only on the symptoms and removing them may mislead the developer and the program may still have some concurrency errors.

Appropriate correctness criterion and bug fix: The correctness criterion formalizes the properties that must be satisfied by the program so that they execute without unintended side effects of concurrency.

In the absence of annotations that indicate the developer's intent, such as atomicity or refinement annotations for code blocks, tools are forced to focus on low-level symptoms such as races. This makes it labor intensive to identify the root cause of and a fix for the bug. Even worse, it misleads the developer to treating the symptom rather than the cause. We studied this in [4] and showed this fact on a real example. In that example that we will present in Section 5.1, there are some reported races. Eliminating these races by synchronizing code blocks introduces other problems and the developer has to synchronize the code by using finer-grained atomicity in the program.

One main functionality of the LP Coverage Metrics is to serve as a link between the developer's intent and the behavior of the program. Since the metric works on the real test program and reports the coverage gaps to the developer, it assists the developer in modifying the test programs according to the intended behavior of the

program.

The LP coverage metric aims to find high-level concurrency errors. Because it is designed based on a pattern of a real concurrency bug that leads to concurrency errors, exercising all LP pairs increases the chances of detecting a high-level concurrency error. Refining the code using the LP coverage reports to test the intended behavior makes the developer to think about the atomicity of the code blocks in detail. In short, using the LP coverage metric makes the testing process interactive and enables the developer to think about the program in more detail and refine her intent.

Chapter 3

TESTING AND COVERAGE METRICS

We introduced several approaches for checking the correctness of multithreaded programs in Section 2.4. Some of these techniques, such as dynamic monitoring, can be classified as a testing technique but most of them are under the *formal methods* category. Even though formal methods are gaining more importance in verification of concurrent programs, testing is still a widely used method for checking the programs for concurrency errors. Coverage analysis is one of the basic techniques to increase the quality of testing. In this chapter, we will explain how coverage can be used for improving testing. We will also emphasize that testing criteria of sequential programs are different from testing criteria for concurrent programs. Finally, present a survey about coverage metrics developed for concurrent programs.

3.1 What is Coverage Used For?

Testing is the process of evaluating the quality of a software by running experiments on it. Tests are evaluated according to some criteria called specification and conformance to this specification is the base for the evaluation process [53]. Testing is one of the main steps of software development cycle which consists of testing, debugging and modifying the code [54]. A basic classification can be made according to the abstraction of the testing process: black-box and white-box testing. In black-box testing, the tester does not know about the implementation of the program and she tests the functionality of the program based on input-output relations. White-box testing requires the user to know how the code is implemented and run the tests according to the implementation details. The LP coverage metric is useful for white-box testing like almost all coverage metrics. There is a basic reason for this fact:

since our coverage measurements are based on accesses to shared variables, the tester should know where the variables are accessed in the source code of the program.

Testing is a labor-intensive process and occupies a long time of the software development process. This makes the automation of the process very beneficial [6]. One other automation technique is *test generation* [53]. Test generation is usually the process of generating test cases from high-level specifications. Test generators take the program and the test adequacy criteria as their inputs and generate test cases according to this input [55]. A technique that can be used to determine specifications for testing by test case generators is measuring test coverage, which also can be used to assist the testing process itself.

Coverage metrics are set of requirements that a test program has to satisfy in order to measure the sufficiency of the testing. Measuring test coverage assists developers in several ways. The key usage of coverage metrics is the identification of noteworthy features and behaviors of the program under test that have not been exercised yet. Coverage gaps provide a means by which to improve the quality of testing. Addressing coverage gaps guides test generation towards scenarios and behaviors that may have been overlooked during testing. The creation and examination of the additional scenarios increases the chance of finding bugs which are difficult to trigger. Measuring coverage can also be used to decide to stop testing or simulation. This helps developers to manage and estimate project progress and to determine whether an adequate amount of testing has been performed. Coverage measurement also assists developers and testers to pinpoint what important execution feature a test case examines and reduces the cost of testing by identifying redundant test scenarios in regression tests.

3.2 Coverage Metrics for Sequential Programs and Concurrency

3.2.1 Coverage Metrics for Sequential Programs

Structural testing specifies testing according to the coverage of a set of testing requirements and is the most commonly used testing method [54]. Many coverage metrics have been developed and proposed in the literature for sequential programs. Metrics

for structural testing of a program can be classified according to the flow graph of the program. In the flow graph of a program, the nodes represent the statements or a block of statements in the program. The edges of the flow graph correspond to the execution of a statement or execution of the code block (transition). If there is more than one edge out of a node, it is a branch statement [7]. The most popular metrics for structural testing according to the flow graph of the program are line (statement) coverage, branch and path coverage [54].

Line (statement) coverage: Line coverage criterion requires that each statement in the code should be executed by at least an execution trace of the program. If some part of the code is not executed, this particular code block may be *dead*, which means that it can never be executed. The existence of a dead code block may signal a programming mistake. Thus, the developer wants to execute all statements in the program and if some statements are never covered it is either dead code or more tests are needed to execute them. Note that line coverage is a very weak requirement and is usually used as a starting point for testing.

Branch coverage: A stronger requirement than line coverage. In order to achieve 100 % branch coverage, all nodes in the control flow graph should be visited. Thus, it also implies line coverage.

Path coverage: Branch coverage is still not strong enough to capture all transitions in the flow graph. Suppose that a node in the flow graph is reachable from two different paths. Visiting this node only through one of the paths is enough to satisfy branch coverage. Reaching this node through a different path may lead to an unwanted action. Path coverage extends the branch coverage as a stronger requirement. The path coverage is satisfied by visiting all paths from the beginning node through the end node of the control flow graph of the program [54].

Note that as the requirement gets stronger, its practical feasibility diminishes. Path coverage becomes infeasible for large programs because there may be a very large number of paths or even may be infinite number of paths on the control flow of a large program.

There are other classes of metrics which are based on the data-flow of the program or those which are not for structural testing, such as fault-based techniques which measures the quality of the testing with the ability to find errors. Let us introduce one of the metrics based on the data-flow of the program because we will refer to it later.

All definition-use path (DU-Paths) criterion: This criterion is satisfied if each path that can reach to a variable x in the program should be a subset of a path in the test [54].

This requirement is a very strong one and is infeasible for many programs.

3.2.2 Testing Concurrent Programs is Different

Even though most of the introduced coverage techniques have been proved to be useful for testing sequential programs, they may not be very helpful for detecting concurrency related bugs. Testing concurrent programs is different from testing sequential programs in a number of ways. A sequential program produces the same results when given the same input vector. On the other hand, there is a non-determinism in scheduling of threads in concurrent programs even for tests with the same input vector. During the execution of a program, the thread scheduler may stop running a thread and may schedule a different thread. The choice of picking a thread is non-deterministic and the output may vary depending on the selection of a particular thread and on scheduling points. This makes the set of interleavings for concurrent programs very large and testing all interleavings is almost impossible [53]. Also, in most cases, running the same test suite many times does not create different executions because almost all runs may follow the same execution order and especially rarely observed scenarios that may cause an error may be missed during testing [47].

Differences in testing concurrent and sequential programs necessitate creating different adequacy criteria for concurrent programs' testing. Sequential code coverage metrics such as line coverage or branch coverage are not useful in detecting errors in concurrent programs. Let us illustrate this point on the motivating example pre-

sented in Section 2.3. In this example, if `append` method and `setLength` methods are executed sequentially as shown in Figure 3.1, 100 % line coverage of these methods is achieved. Similarly, 100 % branch coverage of these methods can be achieved with the execution shown in Figure 3.1 because two conditions of the branch statement (`if` statement) are executed. However, since the interleaving that creates the bug explained in the motivating example presented in Section 2.3, does not occur when two methods are executed sequentially, we cannot detect the bug.

3.3 Existing Coverage Methods for Concurrent Programs

Even though there is significant work on the adequacy of sequential testing, there is not enough work on the coverage of testing concurrent programs. Yang et. al. proposed a coverage metric in [56] for parallel programs by extending *All (DU-Paths) criterion* for sequential programs which we defined in Section 3.2. Their method is based on building a control flow graph of parallel programs by including communication messages and including the paths created by messages between parallel processes.

Taylor et. al. proposed a coverage criteria for Ada programs in [7]. Their technique is based on rendezvous-style communication and they propose 5 levels of hierarchy of criteria for concurrent programs based on concurrency states and concurrency state graph. A concurrency state consists of states of each process of the system. The concurrency graph consists of nodes and edges where nodes represent concurrency states and edges represent state transitions.

- The *all-concurrency-paths* criterion is at the top of the hierarchy. This is a very strong criterion and it requires to visit every path, including paths of infinite length in the concurrency state graph. A loop is an example of an infinite path. Since concurrency state graph includes all concurrent states and all paths of processes, it is practically infeasible target.
- The second criterion of the hierarchy is the *all-proper-cc-histories* criterion. The *all-proper-cc-histories* requires to visit each path in the concurrency state graph

t1	t2
-----	-----
invoke sb.length();	--
int len = sb.length();	--
int newcount = count + len;	--
if(newcount > value.length)	--
invoke sb.getChars();	--
sb.getChars(0, len, value, count)	--
count = newcount;	--
return this;	--
--	invoke sb.setLength();
--	sb.setLength(0);
invoke sb.length();	--
int len = sb.length();	--
int newcount = count + len;	--
if(newcount > value.length)	--
invoke this.expandCapacity();	--
expandCapacity(newCount);	--
sb.getChars(0, len, value, count)	--
count = newcount;	--
return this;	--

Figure 3.1: The execution trace that hides the concurrency bug but accomplishes 100 % line coverage

which are of finite length. This criterion is still impractical because it still visits each path and state in the program. To make it more practical, they weaken their criterion to visit only states of the program.

- The *all-edges-between-cc-states* requires that each edge that changes the state of the concurrent program should be executed.
- The *all-cc-states* concurrency state coverage criterion states that each state in the program should be visited at least once.

As it can be observed easily, even achieving 100 % coverage for the lowest level criteria is possible using exhaustive state space search. This is very impractical for most industrial-scale software. Because of that, they proposed a weaker criterion based on the communication between parallel processes. The *all-possible-rendezvous* criterion requires that each state that involves in a rendezvous communication should be visited along a path in the program. The authors explain that their method is not suggested to be an ideal set of criteria, but to be used as a starting point for further research in developing a set of adequacy criteria for testing of concurrent programs.

Factor et. al. study the same problem in [8]. In their work, a concurrent program is defined as a system consisting of several processes. The parallel processes communicate via communication channels using *send* and *receive* messages. Thus, their concurrency model is also a representation of rendezvous-style communication system. They propose two sequential and two concurrent test criteria for adequacy of testing.

1. **Statement Coverage:** This is the same requirement for sequential programs as discussed in Section 3.2. The requirement for concurrent programs is that each *send()*, *receive()* statement of each process of the concurrent system should be executed by a trace.
2. **Path Coverage:** This is also the same requirement for sequential programs as

discussed in Section 3.2. The only difference is that the control flow graph for this method is only constructed using *send()* and *receive()* statements.

3. **Order Coverage:** This is a newly defined criterion for concurrent programs. It requires to execute each *send()*, *receive()* pair for each channel in the program. The *send()* and *receive()* pairs are extracted using the partial order relation between messages. The partial-order relation is an approximation of the precise relation derived from the source code. Therefore, this partial-order relation may contain a lot of spurious relations which many of them are impossible to cover.
4. **Complementary Order Coverage:** This criterion requires a larger set because it requires to cover each *send()*, *receive()* pair, even if they are unrelated by the partial-order of the program. This criterion targets to capture any hidden dependencies between unrelated events.

In both [7] and [8], there is not an implemented tool which uses their model and they do not explain how to use the coverage to improve the testing. The set of adequacy criteria in both approaches may be very large for large programs and these models may be infeasible to be used for practically-used software.

Edelstein et. al. introduced a tool that uses coverage to improve the bug-finding efficiency built on ConTest, a testing tool used by IBM, in [9]. There are two coverage methods used in their tool as heuristics for increasing the efficiency of the noise-maker tool.

1. The first coverage criterion requires that the execution of a method should be interrupted by a context switch at least once and also each method should be executed without any interruptions at least once. This metric is useful in terms of testing the methods which are aimed to be made atomic. It is a feasible target but it is weak because it can only cover a very small set of the interleaving space which may be very large due to non-determinism.

2. The second coverage criterion requires that each method's execution should be interrupted by a context switch and executing a method of the program. For example, if a thread *T1* is executing method *M1*, a context switch should occur before *M1*'s execution ends and another thread *T2* that runs a method *M2* should take the control. This is a stronger metric to find concurrency bugs.

In [9], Edelstein et. al. developed coverage criteria for generating test cases. For the first criterion, they force context switching in the middle of the execution of a method using *yield()* and by changing priorities of threads. For the second criterion, it is harder to generate automated test cases. The tool determines context switches on-the-fly and if it is possible to cover a pair according to the second criterion, a context switch is forced. They use coverage to increase the efficiency of their noise-maker but the usefulness of the coverage analysis is not experimented with in this work.

Bron et. al. introduced a coverage metric, called *Synchronization Coverage*, for concurrent programs in [57]. They implemented their method using ConTest for C and Java programs. As defined earlier, a lock protected block may be *blocked* or *blocking*. If a thread *T1* executes a block *B1*, which is protected by a lock, and another thread *T2* tries to enter a code block *B2* which is protected by the same lock, *T2* cannot enter this section. In that case, *B1* is said to be *blocking* and *B2* is said to be *blocked*. *Synchronization Coverage* requires to test each protected block to be blocked and blocking. Protected blocks in Java are blocks that are wrapped by *synchronized* keyword and by *locks*. There are more requirements that they introduce to increase the efficiency of the metric:

- *Try-lock* is a concurrency construct that does not block but either a demanding thread acquires the lock or if a thread already holds the lock, it fails to get an already acquired lock. The coverage criterion requires that tests should be run that each try-lock operation *succeeds* and *fails*.
- The *wait-repeated* coverage task requires that a *wait()* is called twice in a single

run of the synchronized block. This ensures that the *wait()* is used in a loop.

- Semaphore-wait coverage task requires to test each call to *Semaphore.acquire()* as blocked and blocking.
- Semaphore-try-wait is similar to *try-lock*, the criterion requires to test it as *succeeded* and *failed*.
- *Notify* coverage task requires that each call to *notify()* wakes up a waiting thread.

The previous work on the coverage of concurrent programs are mostly used for improving the testing of rendezvous-style communication systems. They focus on testing messages sent and received between channels. The LP coverage metric is focusing on shared-memory systems. Edelstein et. al. targets shared-memory systems in [9] but their metric focuses on testing the methods if they are protected by proper locks or not. Similarly, synchronization coverage [57] targets shared-memory systems. In the synchronization coverage, only synchronization variables are checked whereas in the LP coverage metric, accesses to all shared variables are checked. That way, we focus on high-level data races and we force the tests to execute a pattern which is associated with high-level concurrency errors.

Chapter 4

THE LP COVERAGE METRIC

4.1 Concurrent Programs: Syntax and Semantics

4.1.1 Syntax and Semantics

In this thesis, we focus on object-oriented languages which access data structures concurrently. In order to model a Java program, let a program P be a representation of the program at Java bytecode-level. Each line of the bytecode is extracted from a source code line of this program. Note that each source code line may correspond to more than one bytecode line. P consists of classes, methods and locations. Formally $P = \langle C, M, L \rangle$ where C represents the classes in P , and M represents the methods of all classes in P . L denotes the set of locations of P where a *location* represents a bytecode instruction of the program. $sr(l)$ function returns the line of the location l on the source code.

Let Ψ represent an execution of P . A *variable* is either an instance field, a static field or an array element. V is the set of all variables in Ψ such that $V = V_S \cup V_L$ where V_S is the set of *shared variables*, the variables that are accessed by more than one thread during an execution Ψ . V_L is the set of *local variables* which are accessible only by the thread it is created by. Each element of V_S is denoted by v_s . An instance field variable is a pair (o, f) where o is an object and f is a field of $cls(o)$. Similarly, a static field variable, (c, s) , consists of a class and its static field. a is an array object where an *array variable* is an element of a , denoted by $a[i]$ such that i is an index of the array a . Formally,

$$V_S = \{(o, f) \mid o \in O \wedge f \in F_{cls(o)} \wedge (o, f) \text{ is shared}\} \cup \{(c, s) \mid s \in$$

$$S_c \wedge (c, s) \text{ is shared} \} \cup \{(a[i]) \mid a \text{ is shared}\}$$

T represents the set of threads created during the execution of a Ψ . T consists of threads t_x where x represents the unique thread id at runtime during the execution Ψ . V represents the variables created during the execution of a Ψ .

An *action*, either a **write** to or a **read** of a shared variable, is a triple, i.e. $\sigma = \text{read}(v_s, t, l)$ or $\sigma = \text{write}(v_s, t, l)$, where v_s is the variable accessed by thread t at location l . $\sigma.v$ refers to the variable which the action accesses which is v_s , and $\sigma.type$ refers to the action type (either **read** or **write**). Similarly $\sigma.t$, and $\sigma.l$ refers to the executing thread and the location of this action respectively. Two actions are *dependent* if they access the same variable and at least one of them is a write action. Note that we are not interested in the values written or read in this thesis because we are looking for the sequence of executions of locations that might create a concurrency error regardless of values of variables. Note that, we define an action as the execution of an instruction on a shared variable and we ignore the executions of other instructions.

A class is a triple $c = (F_c, S_c, M_c)$ where F_c represents the set of *non-static* fields that class c contains. Similarly, S_c is the set of *static* fields of class c . Let O_Ψ be the set of object instances of C created during execution Ψ . $cls(o) = c$ if the object o is type of class c .

A Ψ is a triple consisting of the sequence of actions, threads and variables. Formally, $\Psi = \langle \pi, T, V \rangle$ where π represents the sequence of actions executed during the lifetime Ψ .

We assume that the program is race-free [13]. Thus, there is a *happens-before* relation between any two accesses to a shared variable [13]. Note that, in this thesis we use *happens-before* and *happens-after* interchangeably. If there is a happens-before relation between **a** and **b**, this implies that there is a happens-after relation between **b** and **a**. In a race-free execution, we can derive a *total order*, denoted by $\rightarrow$, between all the actions in a single execution that is consistent with the happens-before relation. The *sequence of actions*, denoted by π , represents the actions in an execution Ψ , of

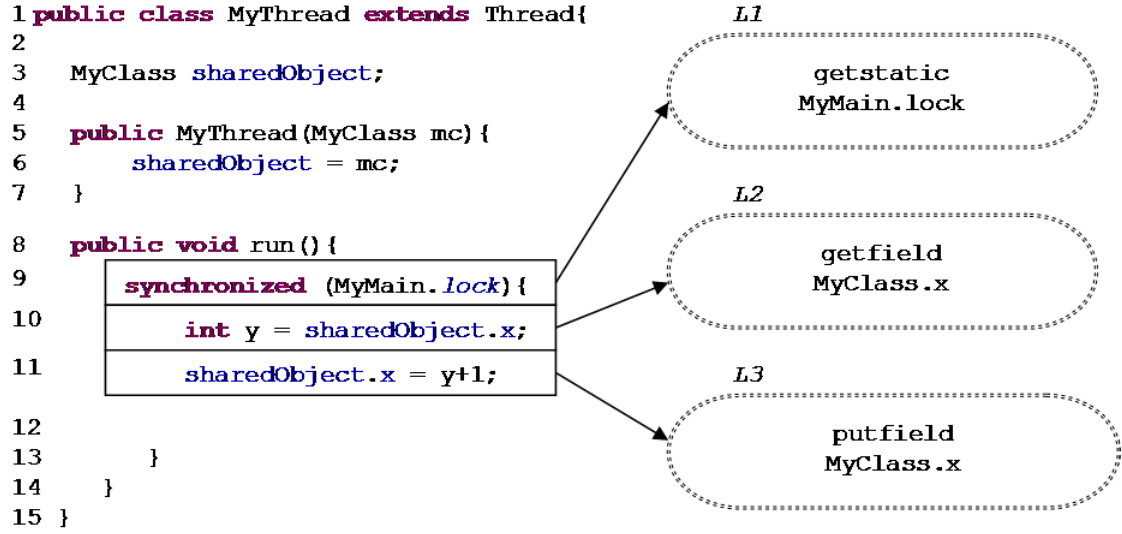


Figure 4.1: Locations in a Java program

P. π consists of a sequence of actions $(\sigma_1, \dots, \sigma_n)$ such that for each pair of actions σ_i and σ_j , $\sigma_i \rightarrow \sigma_j$ where $i < j$. $\sigma_i \rightarrow \sigma_j$ means that σ_i has happened before σ_j in the execution.

4.1.2 Locations and Transitions in Java

We will explain the syntax and the semantics of our concurrent language on a simple Java program. In Figure 4.1 a thread class and its corresponding `run()` method is presented. Each thread has to acquire a common lock before accessing the shared object and running their operations on them.

Let us pinpoint the locations of this program that accesses to shared variables. At line 9 of the source code, the object `lock` is a static field of `MyMain` class and can be reached by `MyMain.lock` expression. Since it is a static object that can be shared by multiple threads, $\text{MyMain.lock} \in S_{\text{MyMain}}$. `x` is an integer field of `MyClass`, so $x \in F_{\text{MyClass}}$. In Figure 4.1, there are three locations. A `getstatic`, which is a bytecode

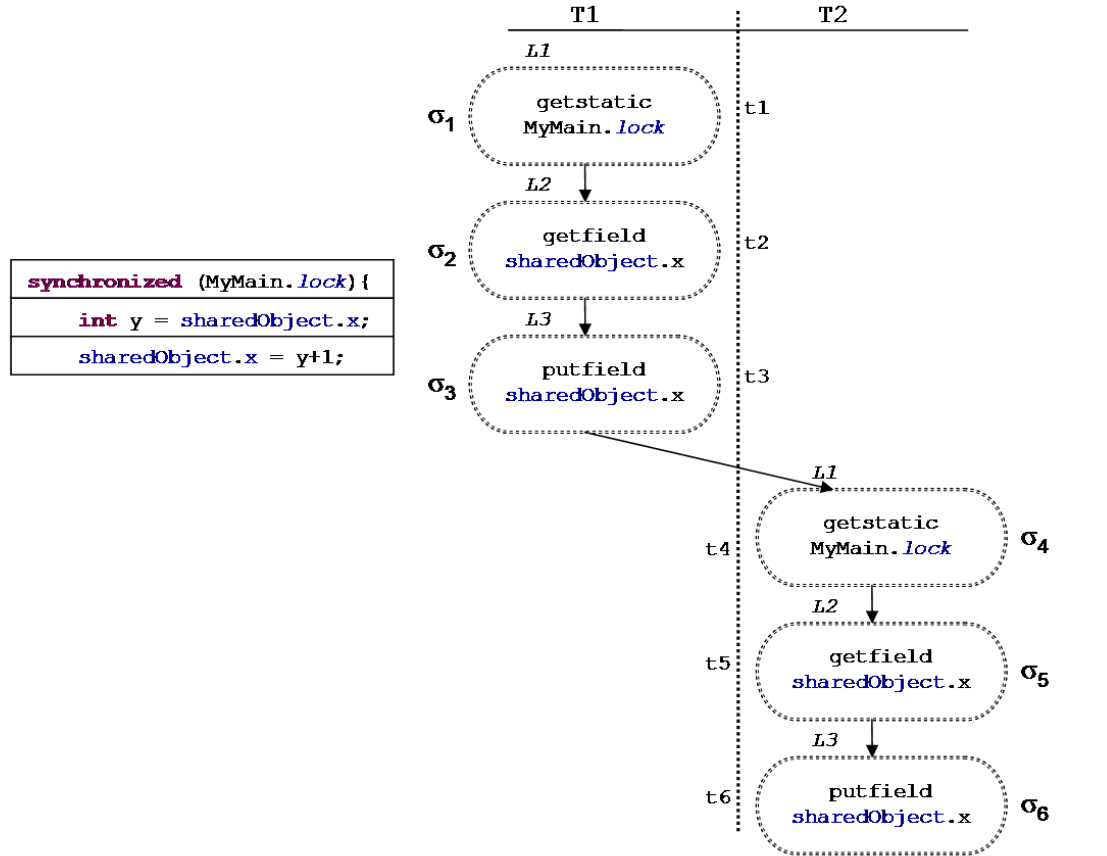


Figure 4.2: Execution trace of a Java program

instruction to read values from static fields, can be executed on `MyMain.lock` and the corresponding bytecode is the $L1$ location of the program. The $L2$ location is the *getfield* instruction, which is a read from the instance field x . The last location, $L3$ is a *putfield* instruction, which a write to the instance field x . Note that the variable $y \in V_L$ and there are no global field instructions running on it. In this example, we use the source code for illustration purposes. Similarly, we use the source code to report the coverage reports to the user. Since we use the program's bytecode for measuring the LP coverage, the compiler optimizations do not affect our results.

Figure 4.2 represents an execution trace of this program. In this example, two threads are running concurrently. Note that the accesses to `sharedObject` is protected

by a common lock and these accesses can be ordered by happens-before relation in the program in dynamic analysis. In this example, the actions of threads are ordered by global time and the actions are presented as a flow graph from top to bottom in their execution order. The difference between the information extracted from the static analysis and the dynamic monitoring is that we know which objects are accessed during the execution while we monitor the program. Let us call the execution shown in Figure 4.2 Ψ_1 . Two threads $T1, T2 \in T$ where T represents the threads created during lifetime of Ψ_1 . $\text{sharedObject} \in O_{\Psi_1}$ and $\text{MyMain.lock}, \text{sharedObject.x} \in V_S$. There are six actions in Ψ_1 :

- $\sigma_1 = \text{read}(\text{MyMain.lock}, T1, L1)$
- $\sigma_2 = \text{read}(\text{sharedObject.x}, T1, L2)$
- $\sigma_3 = \text{write}(\text{sharedObject.x}, T1, L3)$
- $\sigma_4 = \text{read}(\text{MyMain.lock}, T2, L1)$
- $\sigma_5 = \text{read}(\text{sharedObject.x}, T2, L2)$
- $\sigma_6 = \text{write}(\text{sharedObject.x}, T2, L3)$

The happens before relation of Ψ_1 is $\sigma_1 \rightarrow \sigma_2 \rightarrow \sigma_3 \rightarrow \sigma_4 \rightarrow \sigma_5 \rightarrow \sigma_6$. Thus, π of $\Psi_1 = (\sigma_1, \sigma_2, \sigma_3, \sigma_4, \sigma_5, \sigma_6)$.

$\sigma_2, \sigma_3, \sigma_5$ and σ_6 are dependent actions because they operate on the same field of the same shared object.

4.2 The Aim of the LP Coverage Metric

Throughout this thesis, executing a *location*, l , stands for executing the bytecode which builds this location. For example, executing $L3$ on the example presented in 4.1 refers to executing the *putfield* instruction on the variable x .

A *pair* in an execution consists of two locations, let say l_1 and l_2 , that execute dependent actions. Executing a pair implies that if a pair consists of two locations l_1 and l_2 , l_1 must be executed after l_2 . Our test adequacy criterion requires that locations of a pair must be executed by different threads. Therefore, for covering a pair, two locations of this pair l_1 and l_2 , at least one of which executes a write action, l_1 must be executed after l_2 and both locations should be executed by different threads. In the example presented in Figure 4.1, $L2$ and $L3$ constitutes a location pair because they can access the same variable where at location $L3$, there is a write the this variable.

The ordering of the locations is also important. For example, the $(L2-L3)$ pair and the $(L3-L2)$ pairs are different pairs in Figure 4.1. If we have $(L2-L3)$ pair as a coverage target, it implies that the location $L2$ must be executed before the location $L3$.

Here is a demonstration of the usage of the metric on the motivating example presented in Section 2.3: Let the read to `sb` on line 1 of the `append` method be on location l_1 , the read access to `sb` on line 5 of the `append` method be on location l_2 and the write access to `sb` of `sb`'s `setLength(int newLength)` method be on location l_3 . Since l_1 and l_3 are accessing the same variable and l_3 executes a write on the variable, our metric requires to execute l_3 after l_1 without any other locations accessing `sb` and this sequence creates the bug that was explained in Section 2.3.

Throughout this thesis, we use *pair* and *locations pair* interchangeably. Covering a pair (l_1-l_2) means that l_2 is executed after l_1 , there were no other accesses to the field that these locations access and two locations are executed by different threads. The adequacy criterion of the LP metric is to cover all pairs in the program. While presenting the experimental results, we will demonstrate that approaching 100 % LP coverage increases the chance of encountering a high-level concurrency error. We will also show that other design errors such as misuse of synchronization can also be detected by evaluating the LP coverage reports.

method 1	method 2
-----	-----
lock(o){	lock(o){
l1: read from v	l3: write to v
l2: read from v	}
}	

Figure 4.3: An Uncoverable Pair

4.3 Definition of the Metric

There are three main parts of our metric: constructing coverable pairs set, measuring coverage on test programs and reporting coverage to the user.

4.3.1 The Coverable Pairs Set

A *coverable pair* is a pair of locations where the same shared variable can be accessed by these locations and by different threads during some execution where at least one is a write action where no other locations necessarily access the same shared variable between them. In order to illustrate this point, let say l_1 and l_2 read variable v in a lock protected block. Another location l_3 is in a code block which is protected by the same lock and it writes to variable v . The locations are shown in Figure 4.3. The (l_3, l_2) pair is an *uncoverable pair* because l_1 and l_2 are executed atomically and it is impossible to execute l_3 after l_1 is executed and before l_2 's execution is finished. Let us explain this point with an example. In the example shown in Figure 4.1, the $(L3-L3)$ pair is an uncoverable pair. As seen in the example, lines $L2$ and $L3$ are executed in a lock protected block. Also, both $L2$ and $L3$ access the same shared variable of the same object. Thus, it is impossible to execute the location $L3$ after another thread executes this location because before reaching to the location $L3$, $L2$ is inevitably executed.

All coverable pairs constitute the *Coverable pairs set*, denoted by CP , which rep-

resents the set of pairs that are required to be covered at the end of tests. Formally, CP consists of pairs (l_1, l_2) such that the program has an execution with sequence $\pi = (\sigma_1, \dots, \sigma_n)$ that satisfies the following condition:

$$\begin{aligned} \exists v_s, i, j \text{ such that } \sigma_i = \text{read}_s(v_s, t_1, l_1) \text{ and } \sigma_j = \text{write}_s(v_s, t_2, l_2), \ i < \\ j \text{ and for all } k, \ i < k < j \ \sigma_k.v \neq v_s \ t_1 \neq t_2 \end{aligned}$$

We use static analysis to construct an approximation to the set of coverage tasks. Accesses to shared variables are decided at the end of this phase and *coverable set of pairs* is built. Note that it is impossible to exactly capture the relation defined above by static analysis because exact reachability analysis and statically deciding which location can be executed in what order is an undecidable problem. Because of that, we construct a set called *statically detected pairs set*, denoted by SP , which is a larger set than CP and includes CP . SP is constructed by elements which consist of two locations that may access the same shared variable.

For most programs, the number of *uncoverable pairs* in SP is expected to be small and the developer is expected to identify and to take out these pairs as she gets coverage reports. SP is not unique and it very closely depends on the static analysis used. A more effective static analysis builds a more precise SP set that is a better approximation of CP .

4.3.2 Measuring Coverage

The aim of the metric is to detect two locations that execute dependent actions by different threads consecutively. A total order relation was explained in Section 4.1 between the actions of Ψ . A more precise order relation needs to be defined to explain the algorithm well. This relation, *just happened after*, which is denoted by $\xrightarrow{\text{just}}$, means that a location accesses to a shared variable after another location accesses to the same variable and there were no other locations which accesses the same variable between these two locations. Formally;

in an execution Ψ

$$\sigma_i \xrightarrow{just} \sigma_j \Leftrightarrow \sigma_i \rightarrow \sigma_j \mid (\neg \exists k \mid \sigma_i \rightarrow \sigma_k \wedge \sigma_k \rightarrow \sigma_j \wedge \sigma_i.v = \sigma_j.v = \sigma_k.v \wedge i \neq j \neq k)$$

We derive the covered pairs set of an execution from the *just happened after* relation. A *covered pair* is a pair of two *locations* that executes two dependent actions which are executed by different threads. Since we want to extract two consecutively executing locations, there should be a *just happened after* relation between these two locations of a *covered pair*. The set of covered pairs of execution Ψ is denoted by $\Delta(\Psi)$ and formally computed as follows:

$$\Delta(\Psi) = \{(\sigma_i.l, \sigma_j.l) \mid \sigma_i \xrightarrow{just} \sigma_j \wedge (\sigma_i.type = write \vee \sigma_j.type = write) \wedge \sigma_i.t \neq \sigma_j.t\}$$

For example, for the execution trace presented in Figure 4.2, *L2-L3* pair is covered. As shown in the figure, action σ_3 writes to the `sharedObject.x` field at *L3* and the first access to the same field is a *read* action by another thread's action, σ_5 . Since these two actions are executed by different threads and there are no other actions that access the `sharedObject.x` field between these actions, there is a *just happened after* relation between *L2-L3*. So, the (*L2-L3*) is recorded as a covered pair.

4.3.3 Reporting Coverage

When the coverage analysis finishes, the set of pairs that is covered after execution Ψ is recorded into $\Delta(\Psi)$. After a number of executions $\Psi_1, \Psi_2 \dots, \Psi_n$, we get the coverage report of our test programs. Formally;

$$\Delta = \{\Delta(\Psi_1) \cup \Delta(\Psi_2) \dots \cup \Delta(\Psi_n)\}$$

The set of pairs that need to be covered, denoted by R , is reported to the user after the pairs in Δ is extracted from the set SP .

Given a set of test runs, $Test = \{\Psi_1, \Psi_2, \Psi_3, \dots, \Psi_n\}$

$$R(Test) = SP - \Delta$$

Note that the covered pairs set is a cumulative coverage set because as explained before, the concurrent programs may have different interleavings at each run of the

same test. Each run of the test suite may cover different pairs and the cumulative set is reported.

4.4 The Implementation

4.4.1 Static Analysis with Chord

The Algorithm

For constructing location pairs, we use a static race detection tool, Chord, by Naik et. al. [5]. Chord detects potential races by statically operating on Java bytecode. In order to detect potential data races, it detects which memory locations are possibly accessed where in the code, and we use this information to construct our coverage target set. There are 4 phases of Chord. Among these 4 phases, we use the results produced after reachable-pairs computation, aliasing-pairs computation and escaping-pairs computation but not unlocked-pairs computation [5]. As we explained in Section 2.4, reachable-pairs computation computes accesses which are reachable within a running thread and aliasing-pairs computation computes if two accesses are on the same memory location. Finally, escaping-pairs computation decides if these accesses are to a thread-shared variable. We do not use unlocked-pairs analysis which decides if accesses are lock protected. One reason behind that is a potential misuse of synchronization. Locking more elements than needed may harm the performance of the program or even it may cause concurrency errors itself. As a result, we also require to cover protected fields by the test program and the developer may neglect these protected fields when she is sure that the synchronization is used properly. This point will be illustrated in Section 5.1 on a real problem. Also, we want to see the interesting interleavings and we want to see if the accesses to shared-variables are lock-protected or not. If the developer is sure that the accesses are protected by proper locks, she can take out some pairs which are impossible to cover.

The set of pairs that need to be covered is CP , which is explained in Section 4.3. This set is an idealized set and it is an undecidable problem to construct the exact

CP set using static analysis. Therefore, we construct the set called SP set using the output of static analyzer Chord. SP is a larger set than CP and it contains CP . How well SP approximates CP very closely depends on the static analysis used and the program analyzed.

In our implementation, we first run the static analysis and compute the SP set as shown in Figure 4.4. Let say the SP set constructed using Chord is SP_{Chord} output as follows: Let a *statically detected element*, ω denote an access to a shared variable detected by Chord and Ω be the set of ω 's. ω is either a **write** to or a **read** of a shared variable and is a tuple, i.e. $\omega = read_s(e_s, l)$ or $\omega = write_s(e_s, l)$. e_s denotes an element and is either an instance field, a static field or an array object detected to be accessed. l is the location of that access. $\omega.e$ refers to the *statically detected element* which the action accesses, $\omega.type$ refers to the action type (either read or write) and $\omega.l$ refers to the location of ω . Formally,

$$SP_{Chord} \{(\omega_1.l, \omega_2.l) \mid \omega_1.e = \omega_2.e \wedge (\omega_1.type = write \vee \omega_2.type = write)\}$$

Let us explain the *statically detected elements* of the example shown in Figure 4.1. The static analyzer tool takes the program's bytecode and extracts the access information using the bytecode. There are three *statically detected elements* in that example. The read from the static field extracted to be on the location $L1$ is ω_1 , the read from the instance field extracted to be on the location $L2$ is ω_2 , and the write to the instance field extracted to be on the location $L3$ is ω_3 :

1. $\omega_1 = read_s(MyMain.lock, L1)$
2. $\omega_2 = read_s(MyClass.x, L2)$
3. $\omega_3 = write_s(MyClass.x, L3)$

After parsing the results of Chord, a set Ω_e is created for each element e which is either an instance field, a static field or an array object. Then each access ω to an element e is saved in its associated set Ω_e . To construct SP_{Chord} , we traverse each Ω_e

Static Analysis(P)

```

1   $\Omega$  is constructed after static analysis results are parsed
    $\Omega = \{\omega_1, \dots, \omega_n\}$ 
2  for each  $e \in \Omega$ 
3      do
4          for each  $\omega_i \in \Omega_e$ 
5              do
6                  if  $\omega_i$  is write
7                      then
8                          for all elements in  $\Omega_e$ 
9                              do
10                                  $\omega_j = \text{next element in } \Omega_e$ 
11                                 if  $(\omega_i, \omega_j)$  is not in  $SP$ 
12                                     then
13                                          $SP.add(\omega_i.l, \omega_j.l)$ 
14                                          $SP.add(\omega_j.l, \omega_i.l)$ 

```

Figure 4.4: Constructing Coverable Pairs Set

and whenever we find a location $\omega_1.l$ with type **write**, we create a *coverable pair*, cp with $\omega_1.l$ and each other location $\omega_2.l$ in Ω_e . Note that if a pair $(\omega_1.l, \omega_2.l)$ is saved into Ω_e , the $(\omega_2.l, \omega_1.l)$ pair must be saved to the set too. Presence of the $(\omega_2.l, \omega_1.l)$ pair in SP indicates that order of accesses to $\omega_1.l$ must be followed by $\omega_2.l$ to be covered during coverage measurement. We want the pair to be executed in the reverse order too, and because of that the $(\omega_2.l, \omega_1.l)$ pair also needs to be added to the set.

An SP set of size 3 can be constructed using the static analysis algorithm from the example presented in Figure 4.1. As explained above, there are three *statically detected elements* extracted from the source code of this example. Since there is only read on variable `MyMain.lock`, this variable constitutes no pairs. The three pairs are:

1. $(\omega_2.l, \omega_3.l) = (L2-L3)$
2. $(\omega_3.l, \omega_2.l) = (L3-L2)$
3. $(\omega_3.l, \omega_3.l) = (L3-L3)$

So $SP = ((L2-L3), (L3-L2), (L3-L3))$ for the example presented in Figure 4.1.

Pruning the Static Analysis Results

In Section 4.3, we explained what an *uncoverable pair* is and we have shown an example of an uncoverable pair on the example shown in Figure 4.1. We also showed that there is only one coverable pair in the example presented in Figure 4.1, but as explained above there are three pairs in its *SP set*.

Even though it is an undecidable problem to construct the exact *CP set*, there can be refinements on *SP* to decrease the number of uncoverable pairs. Assume that, in our program we can label some code blocks as lock-protected blocks. Lock-protected blocks are blocks in which before the first time a variable is accessed, its object's lock is acquired and held until the last access to it. B is the set of lock-protected blocks, $B = \{b_1, \dots, b_n\}$ where b_x is a sequence of *statically detected elements* that accesses the same field or array object and in the same lock-protected block with their lock is acquired. Function $first(b_x, e)$ returns accesses to e in lock-protected block b_x that might be executed first among other accesses to the locked object's variables. Similarly, $last(b_x, v)$ returns accesses to element e in lock-protected block b_x that might be executed last. $lock(b_1)$ returns the lock that protects lock-protected block b_1 . *isProtected* function returns an access such that;

$$isProtected(\omega_i) = true \Leftrightarrow \exists b_k \in B \text{ such that } \omega_i \in b_k$$

Let say the improved set is called SP_{imp} .

$$SP_{imp} = \{(\omega_1.l, \omega_2.l) \mid (isProtected(\omega_1 = true) \wedge isProtected(\omega_2 = true)) \wedge \omega_1.e = \omega_2.e \wedge \omega_1 \in b_1 \wedge \omega_2 \in b_2 \wedge last(b_1, e) = \omega_1 \wedge first(b_2, e) = \omega_2 \wedge lock(b_1) = lock(b_2)) \vee ((isProtected(\omega_1 = false) \vee isProtected(\omega_2 = false)) \wedge \omega_1.e = \omega_2.e \wedge (\omega_1.type = write \vee \omega_2.type = write))\}$$

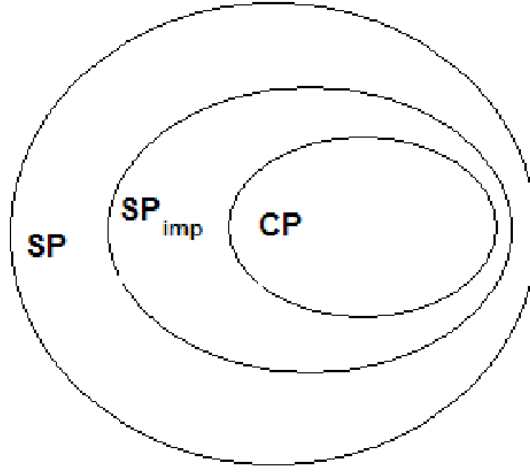


Figure 4.5: The relation of sets

Intuitively, if a location l_1 is the last element of an lock-protected block protected by lock l , and a location l_2 is the first element of an lock-protected block protected by the same lock l where they potentially access the same field of the same object or same element of an array object, we want to add only pair (l_1, l_2) to the SP set. Otherwise, if the access is not inside a lock-protected block, we require its location to be executed after each location which accesses to the same element. Let us call the improved SP set SP_{imp} . The relation between these sets are shown in Figure 4.5. CP is the ideal set of pairs to be covered and SP is the set of pairs constructed by static analysis. The tester can approximate the CP set by improving the SP set and running the coverage based on SP_{imp} will lead to get more precise results.

In the example presented in Figure 4.1, it is easy to identify lock-protected blocks and uncoverable pairs because the code is very small. The developer can easily take out the $(L2-L3)$ pair and the $(L3-L3)$ pair by inspecting the code. For large programs it is not easy to see the atomicity relation of the code blocks. We will propose a solution for deciding uncoverable pairs by annotating code in Section 6.2.

4.4.2 Runtime-monitoring with JPF

We implemented the Location Pairs Coverage Metric on a popular model checking tool JPF [25] using Java. JPF executes Java programs in possible different execution paths and checks the state of the program to detect deficiencies. In a model checker, it is important to deal with the state space explosion problem and to produce a counter example to the developer. In order to do that, JPF saves program states to check whether the current state was visited before and saves the trace of the program at each state to report the path that causes the error to the user. Saving each new state and saving the path to every state by the model checker cause excessive usage of the memory. JPF has a VM which executes Java bytecode and in the LP coverage metric, we used JPF only as an instrumented virtual machine which records some actions. Because of that, we turned off all state and trace caching functionality to eliminate capacity problems. This is harmless to our implementation because we do not need to backtrack to a previously saved state or report the trace of the current state. We measure coverage of test programs and it is sufficient for us to monitor the normal execution of the program. This way, we can handle large programs using limited amount of memory in reasonable time.

After constructing the *SP*, we run the *coverage measurement* algorithm. We implemented the coverage measurement algorithm as a Virtual Machine Listener, *VMListener*, as shown in Figure 4.6. *VMListener* is an interface in JPF which is used to observe the actions taken by JVM. We use `instructionExecuted(JVM vm)` method of this interface which registers the last executed instruction and enables us to analyze the instruction. Since we only check for accesses to shared variables, we are interested in the instructions that access the memory in Java which are listed as follows:

- *Read from static variables:* GETSTATIC
- *Write to static variables:* PUTSTATIC

- *Read from instance variables:* GETFIELD
- *Write to instance variables:* PUTFIELD
- *Read from arrays:* ArrayLoadInstruction
- *Write to arrays:* ArrayStoreInstruction

We check each instruction if it is one of the instructions above. Whenever an instruction which accesses the memory is executed, we create an action σ with the variable that is accessed, v , of type read or write, its thread information, ti , and location l . The last action encountered is denoted by σ_c . In order to determine whether there is a *just happened after* relation between two actions, we use a stack of size 2 for each variable, i.e. if $\sigma.v = \mu$, all actions on variable μ is saved to stack $VStack_\mu$. If the stack is already full, the first pushed element among the two is deleted and new action is pushed on top of the stack. Thus, if there are two elements in the stack, there is a just happened after relation between these two actions. We then check the two actions to decide if they build a covered pair. If at least one of the actions is of type write and these actions are executed by different threads, the locations l_1 and l_2 that executes the two saved actions in the stack is recorded as a covered pair.

An example run of the dynamic monitoring algorithm is presented in Figure 4.7 for the execution shown in Figure 4.2. In this Figure, the leftmost column presents the execution trace of two threads which is also presented in Figure 4.2. The second column presents the contents of the stacks for each accessed variable in the program. The third column is the action which is taken upon an execution of a particular location at the execution trace. For example, at time $t1$, `MyMain.lock` variable is accessed and since there was no stack associated with this field, a stack $VStack_{MyMain.lock}$ is created for this object. Whenever an access to a variable is detected, its associated set is updated accordingly. For example, at time $t5$, σ_5 accessed `sharedObject.x` and $VStack_{sharedObject.x}$ is updated. The content of $VStack_{sharedObject.x}$ was (σ_2, σ_3) prior to that access and it is updated to (σ_3, σ_5) after the access because the last access to

Coverage Measurement(Ψ)

```

1  instructionExecuted(JVM vm){
2
3  insn = last executed instruction
4  ti = the thread that executed the instruction, insn
5  l = location of insn
6
7  if insn is an access to memory
8      then
9           $v$  = the variable that insn reads or modifies
10         type = type of insn, either a read or a write
11          $\sigma_c$  = type( $v$ , ti, l)
12         if  $VStack_v$  does not exist
13             then
14                 create  $VStack_v$ 
15                  $VStack_v$ .push( $\sigma_c$ )
16         if  $VStack_v$ .size = 2
17             then
18                  $\sigma_p$  = Previously inserted element of the stack
19                  $\sigma_c$  = Last inserted element
20                 if ( $\sigma_c$ .type = write or  $\sigma_p$ .type = write) and
                     ( $\sigma_c.t \neq \sigma_p.t$ )
21                     then
22                          $\Delta(\Psi)$ .add( $\sigma_c.l, \sigma_p.l$ )
23 }
```

Figure 4.6: Measuring Coverage

`sharedObject.x` was by action σ_3 . At that point, a covered pair is detected because the threads of actions are different in $Vstack_{sharedObject.x}$ and one of the actions is a write action. As a result, the locations of the actions in this set, $(L3, L2)$, is recorded as a covered pair.

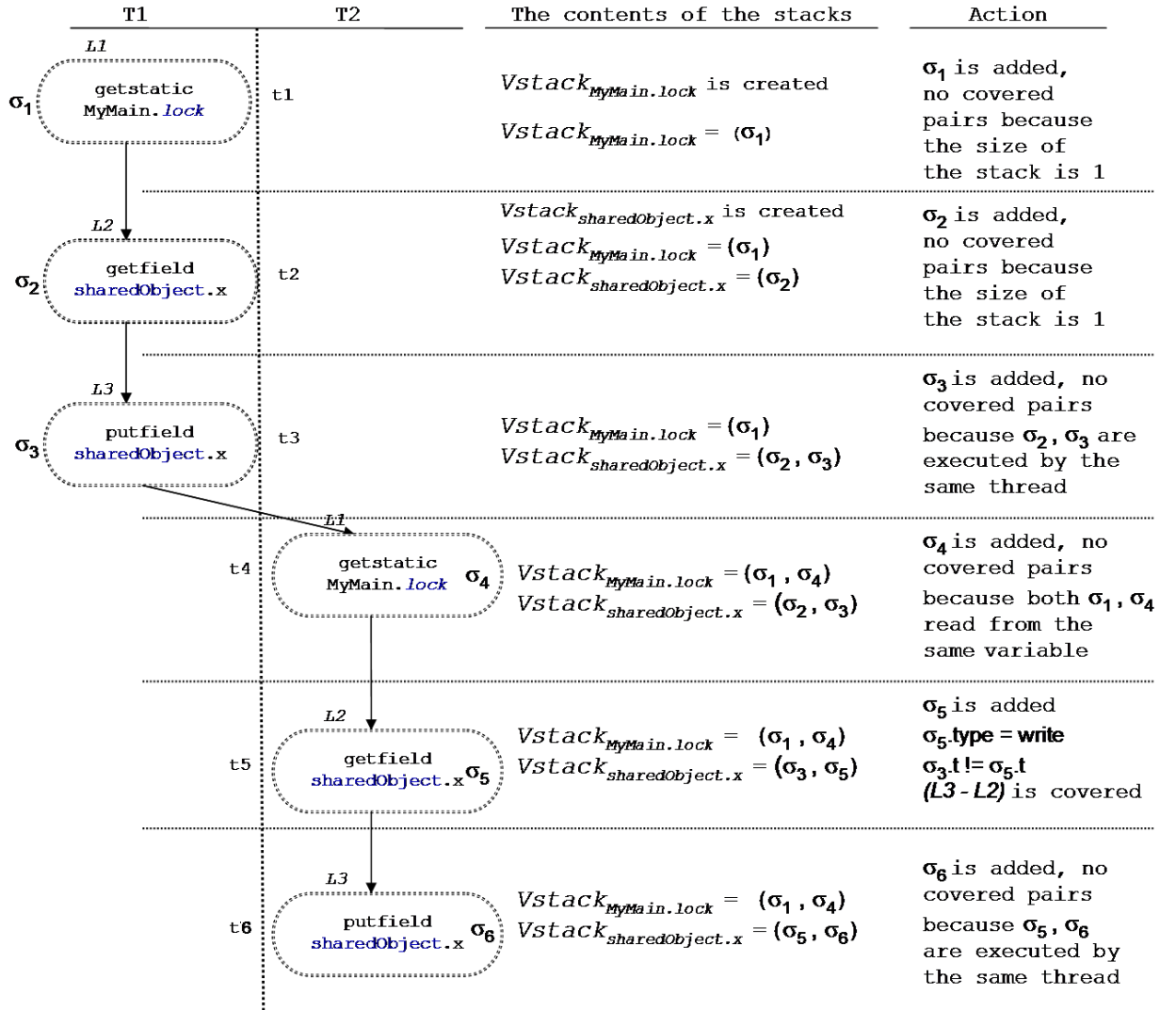


Figure 4.7: The measurement algorithm

```

Coverage Report( $SP, \Delta$ )
1  for all elements in  $SP$ 
2      do
3           $(\sigma_1.l, \sigma_2.l) =$  next element in  $\Delta$ ,
4          if there is a pair  $(\omega_1.l, \omega_2.l)$  in  $SP$  such that
               $\sigma_1.type = \omega_1.type$  and  $\sigma_1.v_s = \omega_1.v_s$  and
               $\sigma_1.l = \omega_1.l$  and
               $\sigma_2.type = \omega_2.type$  and  $\sigma_2.v_s = \omega_2.v_s$  and
               $\sigma_2.l = \omega_2.l$ 
5              then
6                  report  $(\omega_1.l, \omega_2.l)$  as covered and
                      remove from  $SP$ 
7  Report remaining elements in  $SP$  as uncovered

```

Figure 4.8: Reporting Coverage

4.4.3 Reporting Coverage

The *coverage report* phase is shown in Figure 4.8. All pairs covered after each execution Ψ was saved into $\Delta(\Psi)$. After a number of executions, we construct Δ with the conjunction of $\Delta(\Psi)$'s. We traverse all pairs saved in Δ . After getting the next pair from Δ , we compare the both elements' type, variable, location and source line with the elements saved in SP . If there is a statically detected pair which accesses the same variable on the same location, we remove this element from SP and we report this pair as covered. At the end of this phase, all remaining elements in SP is reported as uncovered and all elements in Δ is reported as covered.

Chapter 5

EXPERIMENTAL RESULTS

In this chapter, we present the experiments that we ran on several Java benchmarks. We used 5 various-scale Java programs. We present each experiment's test setup and we explain how to run the LP coverage on these programs. We show how to use the coverage results to modify tests. We also show that coverage metrics for sequential programs are not helpful for detecting concurrency bugs by measuring the line coverage and the LP coverage of the tests.

5.1 Experiments on Apache FTPServer

In this section, we will introduce a real application and show on this program how our metric is useful for finding the high-level concurrency errors explained in Section 2.2. As we will explain on this example, the LP coverage metric assists the developer to test interesting scenarios to catch concurrency errors. We will also show that some solutions for fixing some atomicity problems causes other concurrency errors and the best fix for the bug can be proposed based on the developer's intent as explained in [4]. The LP coverage metric assists the developer to test the intended behavior of the program and proposes the best fix.

5.1.1 Apache FTPServer

The benchmark is a free, open source FTP server, Apache FTPServer, shown in Figures 5.1 and 5.2 which is from one of the benchmarks used by Naik et. al. in [5]. The explained version of the FTPServer contains several concurrency errors including data races and atomicity violations. For experiments on the FTPServer, we created a simulation that mimics the behavior of the server to be able to run in JPF. In this

simulation of the program, we ignored the logging and the user handling mechanisms of the server. There is only the connection and the request handling mechanisms of the server. In our model, connections can be created and they make requests to the server. The server replies to the requests using stub methods and they are empty replies to the connections that do nothing.

The Apache FTPServer runs as follows: upon a new connection is being established between a client and the server, a thread is started in order to handle requests on this connection by using an instance of `RequestHandler`. The corresponding `run()` method of this connection is shown in Figure 5.1. This method has a loop body which listens to incoming FTP requests from the client. In the body of the `run()` method, the fields `m_request`, `m_writer`, `m_reader` and `m_controlSocket` are accessed. The `close()` method is shown in Figure 5.2. When `close()` is called, if the connection is still open, it sets the `m_isConnectionClosed` field of the current `RequestHandler` instance to `true` and sets the `m_request`, `m_writer`, `m_reader` and `m_controlSocket` fields to `null`. the `run()` and the `close()` methods are designed to be run concurrently by different threads. The `close()` method is executed by a thread different from the connection thread such as the time-out thread or shutdown thread.

5.1.2 The Experimental Setup

The static tool, Chord, reports a number of races. As shown in Figure 5.1 and Figure 5.2, there is no locking to accesses to shared variables the in `run()` and `close()` methods. These unprotected accesses to shared fields may cause data races. The race conditions can be eliminated by synchronizing accesses to these fields by, for instance, wrapping each access to the fields `m_request`, `m_writer`, `m_reader` and `m_isConnectionClosed` in a separate code block, synchronized on `this`. As explained in [4], removing the symptom of the bug (the symptom is race conditions in this case), does not guarantee that the errors disappear. We will now explain how the LP coverage metric can be used for detecting high-level concurrency errors in the

```
1  public void run() {
2
3      m_writer.send(220, null, null);
4      m_reader = new MReader();
5      do {
6          m_request.updateLastAccessTime();
7          String commandLine = m_reader.readLine();
8          ... access to m_controlSocket
9          commandLine = commandLine.trim();
10         if(commandLine.equals("")) {
11             continue;
12         }
13         ...
14         m_request.parse(commandLine);
15         if(!hasPermission()) {
16             m_writer.send(530, "permission", null);
17             continue;
18         }
19         ...
20         service(m_request, m_writer);
21     }while(!m_isConnectionClosed);
22
23     if(!m_isConnectionClosed) {
24         conManager.closeConnection(this);
25     }
26 }
```

Figure 5.1: The run method of the FTPServer

```
1 public void close() {
2
3     synchronized(this) {
4         if(m_isConnectionClosed) {
5             return;
6         }
7         m_isConnectionClosed = true;
8     }
9     disconnect(m_request, m_writer);
10    ...
11    FtpRequestImpl request = m_request;
12    if(request != null) {
13        ...
14        m_request = null;
15    }
16    FtpWriter writer = m_writer;
17    if(writer != null) {
18        m_writer = null;
19    }
20    MReader reader = m_reader;
21    if(reader != null) {
22        m_reader = null;
23    }
24 }
```

Figure 5.2: The close method of the FTPServer

program.

During the analysis we assumed that Java libraries are safe and we ignored memory accesses from Java libraries, so we constructed the *SP* set using the fields of `RequestHandler`. After removing the data races, we run the static analysis and the *SP* set is constructed for the `FTPServer` program. Let say the location *l1* is the read from the field `m_writer` in line 16 of the `run()` method as shown in Figure 5.1 and the location *l2* is the write to the same field in line 18 of the `close()` method as shown in Figure 5.2. (*l1-l2*) pair is an element in *SP*. To cover this pair, there should be no access to field `m_writer` between the execution of line 16 of the `run()` method and execution of the line 18 of the `close()` method. Note that the execution of these lines are assumed to be atomic because we run the tests after synchronizing the accesses to these lines to avoid data races. The *SP* set is constructed by pairs derived from read and write instructions on the fields `m_request`, `m_writer`, `m_reader` and `m_isConnectionClosed`. The number of pairs in the *SP* set is 22.

In the initial test phase, we create two connections and run the server. In this setup, there are two `RequestHandler` objects which are created for each connection as shown in Figure 5.3. When `conManager.newConnection(connection);` is called, the `RequestHandler` object (`connection`) is added to the `conManager` and it starts running the `run()` method of this connection which is shown in Figure 5.1.

5.1.3 Detecting High-Level Concurrency Errors

We then start measuring the LP coverage of the server. After running the tests 20 times, no pairs are reported to be covered during the test phase. Next, we increase the number of connections to 4, 8 and 16 and repeat the tests. No pairs are covered with these setups too. There may be a few reasons for this problem. First, some fields are never accessed during tests. Second, the sequence of actions does not lead us to see pairs covered. Debugging the code tells us that our problem is the former one. We can easily observe that the `close()` method is never called and only accesses in the `run()` method are exercised. Following a closer inspection on the server and the

```
public void run() {  
    conManager = new ConnectionManagerImpl();  
  
    for(int i=0; i<2; i++){  
        RequestHandler connection =  
            new RequestHandler(conManager);  
        conManager.newConnection(connection);  
    }  
}
```

Figure 5.3: Creating two connections

connection mechanism, we discover that the `close()` method is called by a `timer` thread. The `timer` thread regularly checks the connections if they timed out and closes the timed-out connections. Since we focused on testing of connection threads, we did not consider an exceptional case such as time-out and we could not see pairs covered. The only way to cover pairs of *SP* is to force the the time-out mechanism to close the connections. For example we need to exercise an (l_1, l_2) pair where l_1 is a write to `m_reader` on source code line 22 of the `close()` method and l_2 is a read to the same variable on line 7 of the `run()` method. When our test program covers this pair, we notice that the program ends with a null pointer exception. The exception is thrown because of an atomicity violation. We notice that the same violation is possible for other fields when `close()` method is executed before `run()` method checks the `m_isConnectionClosed` field and dereferences `m_request`, `m_writer`, `m_reader` in the loop body.

The solution that naturally comes to mind when an atomicity violation is encountered is to synchronize methods which interfere with each other. In our case, the `close()` and the `run()` methods interfere. Synchronizing the actions of both methods on `this` will eliminate the atomicity problem. After modifying the code, we run

the same tests and measure the LP coverage of tests with 2, 4, 8 and 16 connections. We again face the same problem; after 20 tests for each setup, we can observe that no pairs are covered. In the previous case, the source of this problem was that we were not considering an exceptional case. In that version of the code, even though we run the time-out mechanism, we cannot cover any pairs. Debugging the code tells us that the fields of `close()` method is never accessed. This time, we created this problem when we synchronized two methods of `RequestHandler`. Making actions of `run()` and `close()` atomic disables another thread from closing a connection by running `close()` since once a connection thread is started, it does not terminate unless `m_isConnectionClosed` is set to true and it keeps running in the loop. The only way to set `m_isConnectionClosed` to true is to run the `close()` method but since the lock is acquired by the connection thread, another thread can never run it. Note that, in Section 4.4, we explained that during the static analysis phase, we skip the unlocked-pairs analysis. This example illustrates why we are skipping that phase of the analysis. Improper usage of synchronization and locking mechanisms may be a reason of atomicity and refinement violations and we also hope to capture these faults by the LP coverage metric.

5.1.4 Fixing the Bugs

For fixing that bug, the developer has to allow another thread to call the `close()` method concurrently by reducing the granularity of synchronization between `run()` and `close()`. An appropriate fix for the bug is presented in [4] as shown in Figure 5.4. According to this solution, the entire body of `close()` is synchronized since it is called only once and no other thread should be allowed to run it on the same `RequestHandler` object. However, the developer must implement `run()` using more than one synchronized block.

After the fix is introduced, because new accesses to `m_isConnectionClosed` is added to the code as shown in Figure 5.4, the size of the *SP* set grows. There are 28 elements in the *SP* set of the fixed version of the program. On the other hand, the size

```
    declare m_isConnectionClosed field volatile
    ...
1 public void run() {
2     synchronized(this) {
3         if(m_isConnectionClosed) return;
4         else {
5             m_writer.send(220, null, null);
6             m_reader = new MReader();
7         }
8     }
9     do {
10        synchronized(this) {
11            if(m_isConnectionClosed)
12                break;
13            else {
14                String commandLine = m_reader.readLine();
15                m_request.updateLastAccessTime();
16            }
17        }
18
19        synchronized(this) {
20            if(m_isConnectionClosed)
21                break;
22            else{
23                m_request.parse(commandLine);
24                if(!hasPermission()) {
25                    m_writer.send(530,"permission",null);
26                    continue;
27                }
28                ...
29                service(m_request, m_writer);
30            }
31        }
32    } while(!m_isConnectionClosed);
    ...
```

Figure 5.4: The proposed fix for the bug of the FTPServer

of the SP_{imp} set gets smaller. After pruning the SP set, there remains only 5 elements in the SP_{imp} set. Let us illustrate how we pruned the SP set on an example. Note that the `close()` method is synchronized on `this` in the proposed fix. The location $l1$ is the write to `m_reader` at line 22 of the `close()` method shown in Figure 5.2 and the The location $l2$ is the read to `m_reader` at line 14 of the `run()` method shown in Figure 5.4. $(l1-l2)$ pair is an element of the SP set but it becomes impossible to cover this pair when the `run` method was modified as shown in Figure 5.4. As seen in Figure 5.4, the read to `m_reader` is in a synchronized block and if the `close()` method is executed before accessing this synchronized block, `m_isConnectionClosed` is set to `true`. So, when the connection thread enters in the synchronized block in the `run()` method, it will check `m_isConnectionClosed` and break at line 12 without executing the read to `m_reader` at line 14 of the `run()`.

Note that, for FTPServer example, it is impossible to cover all pairs in the SP_{imp} set in only one run of the test. This is because one of the interfering methods is the `close()` method and we need to call this method to cover most of the pairs. We can only cover one pair on a single test because when `close()` method is called the program terminates. Therefore, in order to cover all pairs of SP set, we need to run different tests with different configurations and report all tests' outputs. The five elements of the SP_{imp} are covered by three different test configurations of the FTPServer.

1. The normal execution, by letting the timer to run the `close()` method at any time, covers three pairs.
2. The pair $(l1, l2) \in SP_{imp}$ where $l1$ is the write to `m_isConnectionClosed` at line 7 of the `close()` method (Figure 5.2) and $l2$ is the read to `m_isConnectionClosed` at line 3 of the fixed `run()` method (Figure 5.4). In order to cover this pair, we need to call the `close()` method before the initialization of the connection is completed. We switch to normal scheduling mode and force the `close()` method to run before the connection thread reaches to line 2 of the

fixed `run()` method. After the `close()` method is executed, the connection thread takes the execution control and executes $l2$. This execution covers $(l1, l2)$ pair.

3. The last uncovered pair in SP_{imp} is $(l1, l2)$ where $l1$ is the write to `m_reader` at line 6 of the fixed `run()` method (Figure 5.4) and $l2$ is the read to `m_reader` at line 20 of the `close()` method (Figure 5.2). This pair is covered as follows: The thread that runs the timer is set to be started and close the connection immediately exactly after initialization is complete. So, we let the `run()` method to reach to line 8 of the `close()` method shown in Figure 5.4 and at that point we force a thread switch. The thread that closes the connection takes the control and we force it to call the `close()` method. After the `close()` method is executed $(l1, l2)$ is covered because the last access to `m_reader` was a write by the connection thread at line 6 of the `run()` method, and at line 20 of the `close()` method there is a read to the same variable by another thread.

At the end, we reach 100 % LP coverage in the fixed version of the `run()` method.

As shown in the tests on FTPServer benchmark, avoiding low-level data races may not ensure the correctness of the software. The developer has a higher-level correctness criterion in mind and the LP coverage metric can be used to express the intended behavior of the program. The developer can identify the unforeseen scenarios that have not been tested yet by examining the reports of the LP coverage such as not letting the timer to close the connection or closing the connection before initialization as presented above.

5.2 Experiments on Multiset

In this section, in order to illustrate that the LP metric corresponds with concurrency better, we measure the line coverage and the LP coverage of test programs. As we noted earlier in Section 1.1, running the same concurrent program many times may not help the developer for detecting the bugs. In our tool, we have a random

thread scheduling mode which we think assists the testing programs to create more interleavings. In this section, we also illustrate how random thread scheduling helps the test programs to produce more interleavings.

5.2.1 Multiset and the Experimental Setup

We used the experimental benchmark multiset to compare our metric with the sequential coverage techniques. The multiset is a data structure that implements insert, delete, look up and insert pair operations. **Insert**(x) inserts an element x into the multiset, **Delete**(x) deletes the element x if there is one whereas **LookUp**(x) returns true if the multiset contains element x and **InsertPair**(x, y) calls the insert operation for two elements x and y [3].

We created a test suite consisting of 100 runs. In each run, n threads are created. Each thread picks i operations among insert, delete, LookUp and insertPair randomly and runs the selected operation with a random number as the method argument. For instance, with $n=2$ and $i=10$, two threads are created and these threads call 10 random operations with a random number as the method argument in their **run** methods. This procedure is repeated 100 times. The tests are run for combinations of n values 2, 4, 8, 16, 32 and i values 5, 10, 20. We ran each test with two thread scheduling modes: i) normal thread scheduling ii) random thread scheduling with these values of n and i .

5.2.2 Experiments

The static analysis phase reports 59 pairs for the multiset benchmark. After running the test suite that we explained above five times for each combination of n and i , we realize that at most 29 pairs are covered for any value of n and i . The reason why test programs can cover almost half of the SP set is that all four methods are synchronized, so their execution do not interfere with each other. Let us clarify this point with an example. We will illustrate a case where a pair is decided to be an uncoverable pair on Figure 5.5 and Figure 5.6. Let location l_1 be the **read** to $A[i].valid$ at line 82 of

the `lookUp()` method shown in Figure 5.6, and l_2 be the `write` to $A[i].valid$ at line 69 of the `delete()` method shown in Figure 5.5. (l_1, l_2) is in SP but it is an uncoverable pair because after `lookUp()` accesses $A[i].valid$ at source line 82, even if `delete()` method is executed, it is impossible that the first access to $A[i].valid$ will be at line 69 because line 65 through line 72 of `delete()` method is updated atomically. So, before executing line 69, $A[i].valid$ at line 67 needs to be accessed first. Remembering that there should be a just happened after relation between two actions to cover a pair, (l_1, l_2) becomes an uncoverable pair because it is guaranteed that there will be an access to $A[i].valid$ between the actions which execute these lines.

```

63 public boolean Delete(int e) {
64   for (int i = 0; i < capacity; ++i) {
65     synchronized (A[i]) {
66       if (A[i].elt == e) {
67         if(A[i].valid){
68           A[i].elt = -1;
69           A[i].valid = false;
70           return true;
71         }
72       }
73     }
74   }
75   return false;
76 }

```

Figure 5.5: `delete()` method of multiset

29 of the pairs can be identified as uncoverable pairs like the example presented above. Thus, there remains 30 pairs in the SP_{imp} set. Recalling that 29 pairs were covered by the tests, there is only one pair which is uncovered in SP_{imp} . This pair is

```
78 public boolean LookUp(int e) {
79     for (int i = 0; i < capacity; ++i) {
80         synchronized (A[i]) {
81             if (A[i].elt == e) {
82                 if(A[i].valid){
83                     return true;
84                 }
85             }
86         }
87     }
88     return false;
89 }
```

Figure 5.6: LookUp() method of multiset

the (l_1, l_2) pair such that, l_1 is the write to `A[i].elt` at line 50 of the `InsertPair()` method shown in Figure 5.7 and l_2 is the read to `A[i].elt` at line 81 of the `LookUp()` method shown in Figure 5.6. In order to cover this pair, the if statement at line 48 of the `insertPair()` method should be true and after a thread accesses line 50 and return false, there should be a thread switch. The other thread which takes the control of the execution should run the `lookUp()` method. This certain execution order can be run as follows: Let say the size of the multiset is s . There should be $(s-1)$ elements in the multiset and a thread should run `insertPair()` method and another thread should run `lookUp()` method concurrently. These two threads' execution must be interleaved as explained above to see this pair covered. Another requirement in order to see that pair covered is that `insertPair()` method should first insert an element which is not in the multiset. It then should try to insert the second element and it should fail. This uncovered pair tells the developer that `insertPair()` method is not tested enough to verify all possible scenarios. This way LP metric assists the

developer to write a test code to execute that sequence of actions and test a rarely seen scenario

```
43 public boolean InsertPair(int x, int y) {
44     int i = FindSlot(x);
45     if (i == -1) {
46         return false;}
47     int j = FindSlot(y);
48     if (j == -1) {
49         synchronized (A[i]) {
50             A[i].elt = -1;
51             return false;
52         }
53     }
54     synchronized (A[i]) {
55         synchronized (A[j]) {
56             A[i].valid = true;
57             A[j].valid = true;
58         }
59     }
60     return true;
61 }
```

Figure 5.7: InsertPair() method of multiset

5.2.3 Test Results

Test results for ($n=16$ and $i=10$) are shown in Figure 5.8. The x axis of the graph corresponds to the number of total method calls and the y axis shows the percentage of the test coverage. There are four curves in the graph. *Normal Scheduling LP Coverage*

curve shows how the LP coverage of the tests increases with more method calls in the normal thread scheduling mode. *Normal Scheduling Line Coverage* represents the percentage of line coverage by number of method calls with the normal scheduling policy. Similarly, *Random Scheduling Line Coverage* corresponds to the reached line coverage of the test by number of method calls and *Random Scheduling LP Coverage* corresponds to the reached LP coverage of the test by number of method calls while running in random scheduling mode.

5.2.4 Interpretation of Test Results

We compared line coverage with LP coverage. As it can be observed in Figure 5.8, when 100 % of line coverage is reached, only 40 % of the LP coverage can be achieved. This result illustrates the point we explained at Section 3.2; classical coverage measuring methods are not sufficient to measure the concurrency testing well. Another fact to illustrate this point is that for the same experimental setup, the line coverage in Normal Scheduling and Random Scheduling modes can reach up to 100 % line coverage almost at same number of method calls. We can conclude that thread scheduling does not actually affect how the line coverage grows. This conclusion implies that the line coverage, which is a coverage method for sequential programs, has a weak correlation with concurrency and thread scheduling. These observations show us that line coverage measurements are independent of how many interleavings the test program created, and therefore they are useless in measuring how well the concurrency is tested.

5.2.5 The Effect of Random Thread Scheduling

As shown in Figure 5.8, Random Scheduling can reach to a higher percentage of LP coverage at same number of method calls compared to Normal Scheduling. By deploying the Random Scheduling mode, we do not only measure the coverage of the current testing code, but we also help the testing to produce more concurrent interleavings of the program.

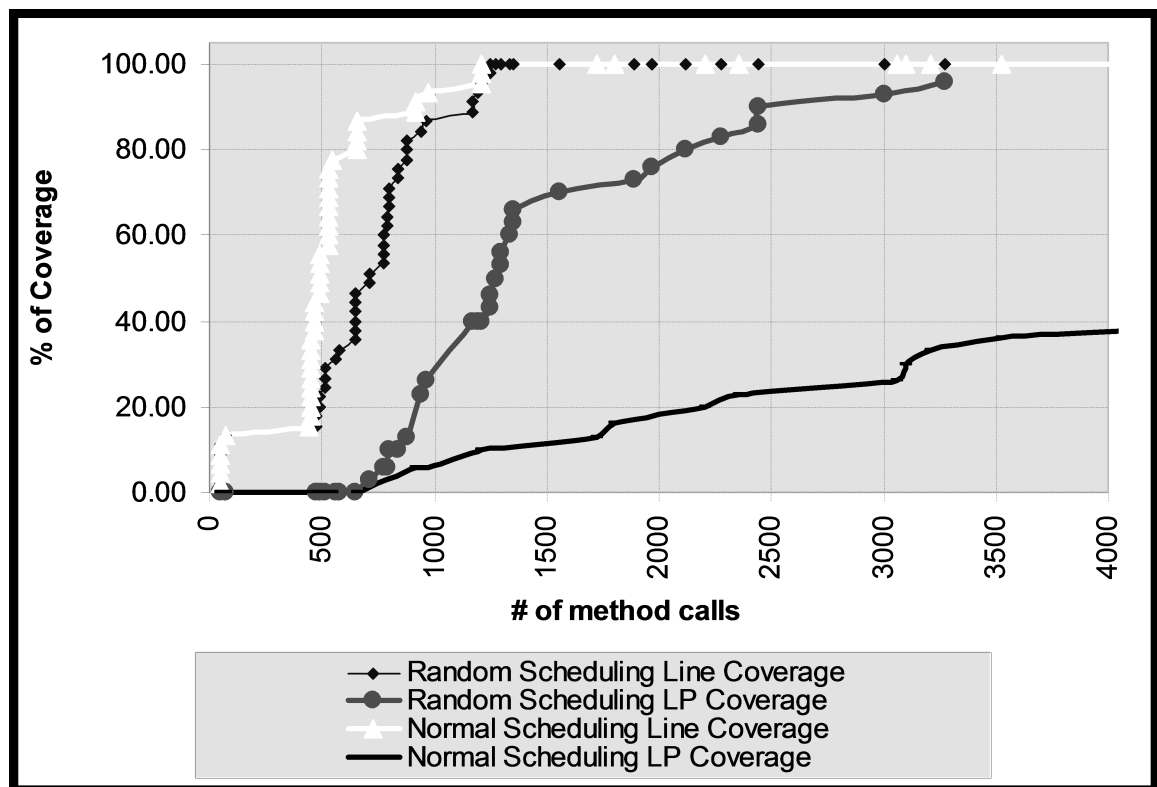


Figure 5.8: Line and LP coverage with 16 threads in Normal Scheduling and Random Scheduling modes on multiset

5.2.6 How to Use the LP Coverage More Effectively

Another conclusion that can be drawn from multiset experiments is that in order to use coverage methods efficiently, the user should know about the program well. At the very least, the intended behavior of the program and the abstract structure of the test suite should be known well. Remember that in the previous benchmark explained in Section 5.1, Apache FTPServer, the number of connections (threads) did not change results of the tests and coverage measurements. In multiset, the number of threads affects test results as shown in Figure 5.9. In the graph presented in Figure 5.9, the x axis corresponds to the number of method calls and the y axis represents the achieved LP coverage. As the number of threads gets larger, more interleavings can be seen and more pairs can be covered in less number of method calls. In multiset, each thread runs the same set of operations concurrently and when we have more threads we increase the chance of creating more interleavings. On the other side, for the FTPServer benchmark, in order to create interesting interleavings, a thread needs to call the `close()` method, which shuts down the connection, and does not allow for covering more interleavings. The tester should be aware of the fact that creating more connections may not help for covering more pairs in the FTPServer program and should modify the program to get more interleavings instead of creating more connections.

In this section, we have shown that the sequential coverage metrics are not helpful for detecting concurrency errors. We also have shown that scheduling threads randomly helps to get more interleavings and increases the chance of catching high-level concurrency bugs.

5.3 Experiments on moldyn

5.3.1 moldyn and Experimental Setup

Moldyn is a computationally intense program which models the interaction of molecular particles [58]. We ran moldyn's own implemented main application by creating

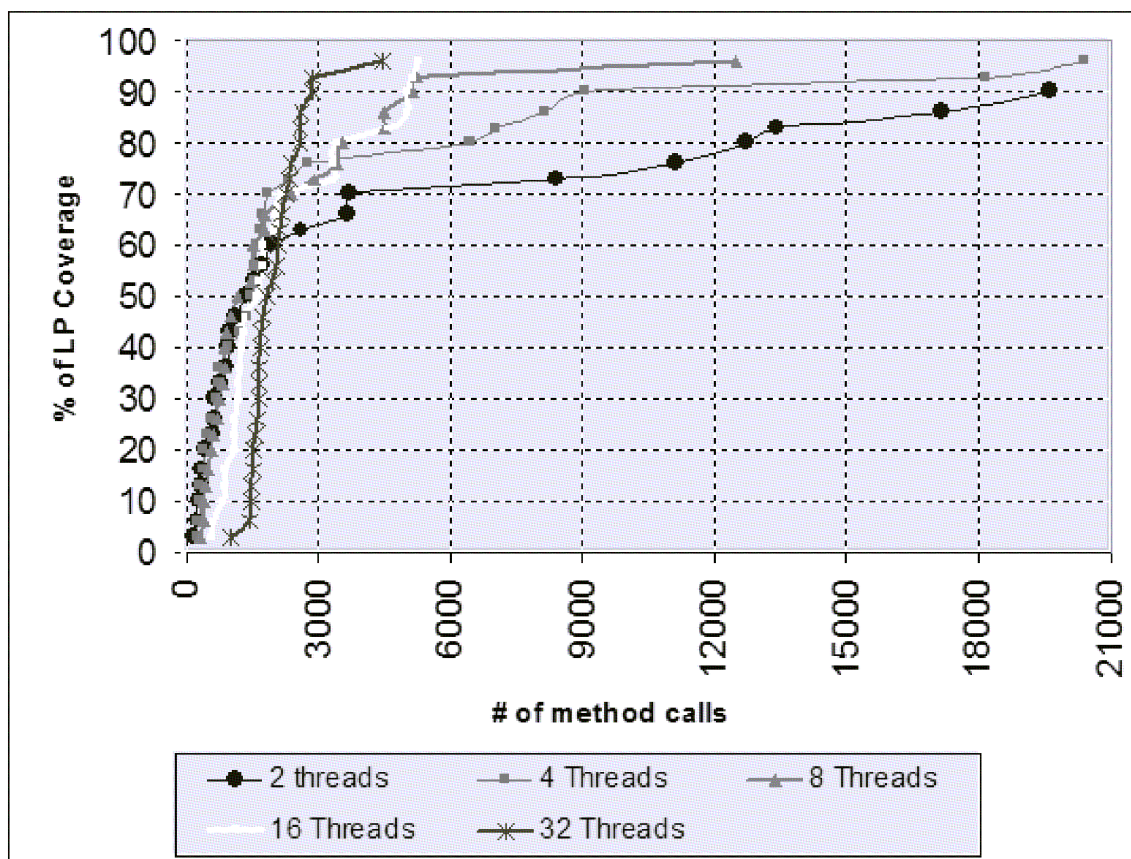


Figure 5.9: LP Coverage with different number of threads on multiset

2,4,8,16 and 32 threads and measured the LP and the line coverage of it. There are a large number of accesses to the shared variables in moldyn because the computations are run in nested loops. Each of these loops has a large number of iterations and this makes the program access each shared variable frequently.

5.3.2 Building the SP_{imp} set

Barrier points are used in moldyn and this assists the developer to construct the SP_{imp} set. Let us illustrate a case where the developer can identify a pair as uncoverable because of the barrier points. A code portion from moldyn is shown in Figure 5.10. Let (l_1, l_2) be a locations pair where l_1 is the write to the static `md.vir[0]` field at line 342 and l_2 be the read to the `md.vir[id]` field at line 406 on the code shown in Figure 5.10. (l_1, l_2) is an uncoverable pair because all threads have to reach the barrier point at line 356 before continuing their execution. Before reaching to this point, the threads have to run the instruction at line 348 and make a write to the field `md.vir[0]`. Therefore, (l_1, l_2) pair cannot be covered because it is guaranteed that there will be another access to the same field between the execution of these locations.

For the moldyn benchmark, there is a total of 26 coverable pairs in the SP_{imp} set.

5.3.3 Experiments

As shown in Table 5.5, when 100 % line coverage is reached, only 34 % of the pairs in the SP_{imp} set had been covered for this benchmark. The graph presented in Figure 5.11 shows how the line coverage and the LP coverage of moldyn increases as the number of method calls grows for 4 and 8 threads. In this graph, the x axis corresponds to the number of method calls and the y axis is the coverage achieved. We have four curves which represents the percentage of line or LP coverage reached by number of method calls for 4 and 8 threads. As shown in the graph, the line coverage can reach up to 80 % even there are no pairs covered yet. Then, the program enters a code region where some interesting interleavings are encountered and it reaches to

```
337  if(id==0) {
338      for(j=1;j<JGFMolDynBench.nthreads;j++) {
339          double d = md.epot[j];
340          md.epot[0] += d;
341          d = md.vir[j];
342          md.vir[0] += d;
343      }
344      for(j=1;j<JGFMolDynBench.nthreads;j++) {
345          double d = md.epot[0];
346          md.epot[j] = d;
347          d = md.vir[0];
348          md.vir[j] = d;
349      }
350      for(j=0;j<JGFMolDynBench.nthreads;j++) {
351          int itr = md.interacts[j];
352          md.interactions += itr;
353      }
354  }
355
356  br.DoBarrier(id);
357
358  ...
403  md.epot[id] = 4.0*d;
404  etot = md.ek[id] + md.epot[id];
405  temp = tscale * ekin;
406  pres = den * 16.0 * (ekin - md.vir[id]) / mdsizes;
407  vel = vel / mdsizes;
408  rp = (count / mdsizes) * 100.0;
409  }
```

Figure 5.10: An uncoverable pair from moldyn

100 % line coverage where the LP coverage can reach up to 34 %.

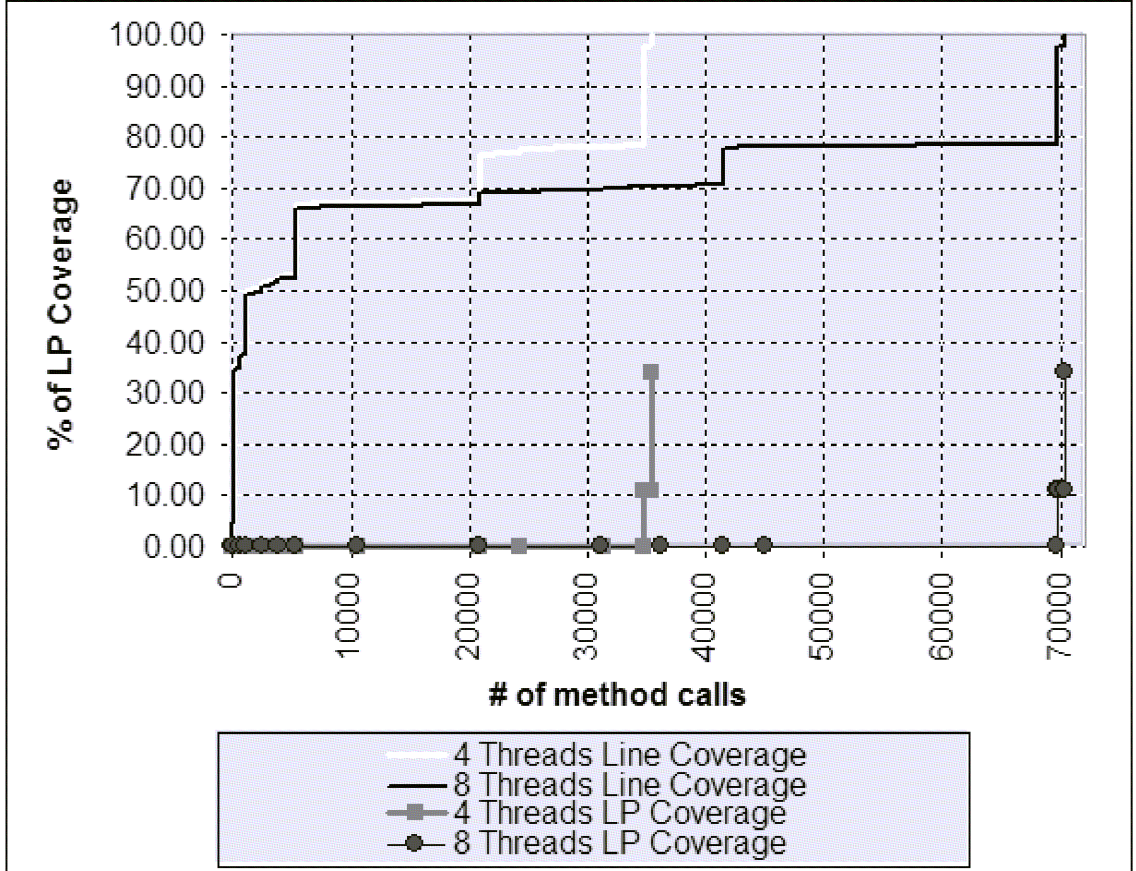


Figure 5.11: LP and line coverage with 4 and 8 threads on moldyn

For the moldyn benchmark, we reached 88 % LP coverage when 2190511 methods calls are made with 32 threads. Repeating the same tests several times does not help to reach 100 % LP coverage. The size of the SP_{imp} set was 26 and we covered 23 pairs of this set. The uncovered three pairs of SP_{imp} are (l_1, l_2) pairs where l_1 is the write to `sh_force[j][i]` at line 361 and l_2 are the read to `sh_force[0][part_id]`, `sh_force[1][part_id]`, and `sh_force[2][part_id]` at lines 552, 553 and 554 on the code shown in Figure 5.12. In the code shown in Figure 5.12, the block between lines 358 and 364 is only accessed by a thread with id 0, because the if statement at line 358 checks the thread id and allow only thread 0 to enter this region. At line 369,

```
358     if(id == 0) {
359         for (j=0;j<3;j++) {
360             for (i=0;i<mdsize;i++) {
361                 sh_force[j][i] = sh_force[j][i] * hsq2;
362             }
363         }
364     }
    ...
369     br.DoBarrier(id);
    ...
    ...
552     xvelocity = xvelocity + sh_force[0][part_id];
553     yvelocity = yvelocity + sh_force[1][part_id];
554     zvelocity = zvelocity + sh_force[2][part_id];
    ...
```

Figure 5.12: A code sample from moldyn

there is a barrier point and all threads wait for each other to reach that barrier point. For all runs of tests, the first thread that was allowed by scheduler to pass this barrier point and executing the lines 552, 553, and 554 was thread 0. Since the lines were executed by the same thread 0, we could not cover these three pairs. After line 369, we pause the thread 0 and let other threads to continue their runs. So, we achieved to cover these remainin three pairs and reached to 100 % LP coverage.

5.4 Experiments on SOR

SOR is contained in Java Grande Forum Benchmark Suite [58]. SOR performs in-tense computationally for successive relaxation on a 2D grid. It was very difficult to construct the SP_{imp} set of this benchmark because the size and the elements of the SP_{imp} set changes with the number of threads and the size of the grid. In our experiments we used a grid of size 50x50. For instance, when two threads are run on the benchmark, there are only 8 coverable pairs and when the numbers of threads is 8, the size of the the CP set is 24. When this fact is ignored and the SP_{imp} is constructed using only Chord output, the size of the set is 96. In such a setting, the percentage of covered pairs was very low and we had to investigate the behaviour of the program. After all we constructed the SP_{imp} sets for 2,4,8,16 and 32 threads. The size of the SP_{imp} is 24 for 4 and 32 threads, 25 for 8 threads, 16 for 16 threads and 4 for 2 threads. Maximum achievable line coverage is 97 % for SOR and the correspondence of line coverage to LP coverage for SOR is shown in Table 5.5. For all settings except with 8 threads, 100 % LP coverage is achieved with test cases. With 8 threads one pair remains uncovered and it can be covered by playing with the thread scheduling.

For the SOR benchmark and the FTPServer benchmark, the line coverage can not reach up to 100 %, because in both benchmarks there are some control statements for exceptional cases, such as the catch branch of a try, which never occured during tests. For the SOR benchmark, we ran the benchmark's own calculation application. As shown in Table 5.5, maximum achievable line coverage is 97 % for SOR and 95 %

```

for (int proc_id = 0; proc_id < nprocs; proc_id++) {
    first_row = (M * proc_id) / nprocs + 1;
    last_row  = (M * (proc_id + 1)) / nprocs;

    if ((first_row & 1) != 0)
        t[proc_id] =
            new sor_first_row_odd(first_row, last_row);
    else
        t[proc_id] =
            new sor_first_row_even(first_row, last_row);
    t[proc_id].start();
}

```

Figure 5.13: Assigning computations at SOR

for FTPServer.

5.5 Comparing the Line and the LP Coverage of Benchmarks

In Table 5.5, we present results for comparing line and LP coverage for benchmark programs. In the first column, we give the name of the program. The second column represents the number of threads that the program runs. The third column is the total number of method calls made by the execution of the program to reach maximum line coverage and the fourth column presents the percentage of the maximum line coverage. The last column is the percentage of the LP coverage when the maximum line coverage is achieved. Note that the LP coverage is measured according to the size of the SP_{imp} set. The last row of this table can be read as follows: for 32 threads running the moldyn benchmark, the maximum percentage of the line coverage is 100 % and it is reached when 280612 method calls were made. When the maximum of line coverage is achieved for that number of threads, 34 % of the SP_{imp} set of moldyn

were covered by the test program. In the table, there are two rows for multiset benchmark, `multiset(i=5)` and `multiset(i=10)`. The value of i corresponds to the number of selected operations at each run of the tests as explained in Section 5.2. For the SOR benchmark and the FTPServer benchmark, the line coverage cannot reach up to 100 %, because in both benchmarks, there are some control statements for exceptional cases which never occurred during tests. For the SOR benchmark, we ran the benchmark's own calculation application. As shown in Table 5.5, maximum achievable line coverage is 97 % for SOR and 95 % for FTPServer. In addition to the experiments explained, experiments about a very small program, *Dining Philosophers* is added to the results.

5.6 Significance of the Experiments

Some important facts can be observed on the experimental results:

1. The LP coverage metric may assist the developer improve the testing of concurrency better by providing coverage gaps.
2. Sequential coverage metrics, such as line (statement) coverage, are not helpful for improving tests for concurrency.
3. The programmer should be familiar with the abstract structure of the program for evaluating the results of coverage metrics.

We also have shown that the LP coverage metric is scalable for realistic configurations of large-scale programs that cannot be model checked or tested exhaustively.

Table 5.5: Experiments on several benchmarks

Programs	# of Threads	# of method calls when max line coverage is reached	% of max line coverage reached	% of LP coverage when max line coverage is reached (% of SP_{imp})
Dining Philosophers	2	84	100	28
	4	132	100	28
	8	144	100	42
	16	138	100	42
	32	180	100	28
Multiset (i=5)	2	979	100	33
	4	1065	100	40
	8	939	100	36
	16	1236	100	36
	32	2012	100	46
Multiset(i=10)	2	1215	100	46
	4	1289	100	40
	8	1079	100	46
	16	1252	100	46
	32	2052	100	56
FtpServer	2	25937	95	40
	4	30149	95	40
	8	43034	95	40
	16	68637	95	40
	32	48843	95	40
Sor	2	80	97	25
	4	140	97	25
	8	248	97	36
	16	462	97	25
	32	907	97	41
Moldyn	2	17954	100	34
	4	35325	100	34
	8	70227	100	34
	16	139995	100	34
	32	280612	100	34

Chapter 6

CONCLUSIONS & FUTURE WORK

6.1 Summary

In this thesis, we presented a coverage metric for measuring the adequacy of testing of concurrent programs. Even though there is quite a little work on measuring the test coverage for sequential programs, there is not much research on measuring the coverage of testing for concurrent programs. The LP coverage metric presented in this thesis, is particularly designed for catching concurrency errors. As we have shown, covering the gaps provided by the Location Pairs (LP) coverage metric helps the developer to find the errors and potential bugs in concurrent programs.

The LP coverage metric is inspired by a common execution pattern that leads to concurrency errors. This pattern of execution creates high-level concurrency errors, such as atomicity and refinement violations and they are harder to detect compared to low-level data races in the programs. These errors may still exist even in the absence of low-level data races and they may be hard to catch with classical testing and verification methods.

We defined a concurrent program based on the bytecode of a Java program. Each atomic bytecode instruction corresponds to a location in the program. The LP coverage metric requires that each location that writes to a shared variable should be executed after and before each other location that accesses the same location. These two locations are recorded as a covered pair if there are no accesses to the same shared variable between the execution of these two locations. The developer can check for high-level concurrency errors by forcing this execution pattern to be tested for each location pair.

We implemented the LP coverage metric and experimented it on several Java programs. The metric has 3 phases: constructing the coverage target set, measuring the LP coverage and reporting coverage to the developer. In the first phase, the set of location pairs is constructed. In our implementation, we used a static race detection tool, Chord. Using the results of Chord, we construct a coverage set but that set is usually larger than the exact coverage. The developer should use the coverage results to prune this set and approximate it to an exact location pairs set to be covered. In the monitoring phase, the execution of the program is monitored and covered pairs are decided. We implemented the monitoring tool using the JVM of model checker JPF. In the last phase, the pairs that are covered and uncovered are reported to the user. The set of covered and uncovered pairs are the conjunction of a number of sets because at each run of the program, there might be different pairs which have been covered.

We measured the line and the LP coverage metric of several Java benchmarks. We have demonstrated how to construct the coverage set and how to use the LP coverage metric to improve testing on these benchmarks. We demonstrated the following facts about the LP coverage metric:

- The uncovered pairs reported by the LP coverage metric helps the developer to test the unforeseen scenarios which may cause to concurrency errors.
- Sequential coverage metrics do not help for improving the testing of concurrent programs.
- Large-scale programs which cannot be handled by model checkers can be monitored by the LP coverage metric.

6.2 Future Work

Automating Pruning of Uncoverable Pairs: In Section 4.4.1, we explained the relations between the CP set, the SP set and the SP_{imp} set. We explained how this

improvement can be useful on multiset in Section 5.2. The user should annotate the lock-protected blocks for this analysis. It is easy in multiset example because the protection mechanism provided by synchronized keyword and all methods are in the same class. Also, there is only one shared variable that is accessed and these relations can be easily seen by the developer. So, for multiset, the developer can construct SP_{imp} by inspection and the result are presented in Section 5.2. Labelling lock-protected blocks and using them to prune the SP set is a future work that will make the tool more efficient by decreasing the number of coverage targets.

Considering Context Sensitivity: In our current implementation, we do not consider context sensitivity. Thus, accessing the same location of a method from different lines is treated equally. For example, if a shared field is only accessed by its `get()` method and modified only by its `set()` method, calling `get()` and `set()` methods anywhere in the code consequently in different orders are enough to cover the pairs that this field constructs. If there are more calls to these methods in the code, they are not required to be run consequently. Treating each call to `get()` and `set()` methods of this shared field as a separate location may help the developer to exercise more interesting scenarios and will increase the possibility of finding a bug if there is one. Let us illustrate the case with an example:

In the motivating example presented in Section 2.3, the error was due to modifying the array length improperly. Now suppose the case we have a method that includes the code block shown in Figure 6.1.

```
synchronized{
    len = sb.length();
    len = len*2;
    sb.setLength(len);
}
```

Figure 6.1: A sample code to show the importance of context sensitive analysis

After running this code block if another thread calls `sb.getchars(...)` method, since we covered the pair constructed by `setLength` $\Rightarrow$ `getchars` method call sequence, there is no need to create a scenario on the motivating example shown in Section 2.3. The code block shown in Figure 6.1 does not create the bug because it doubles the size of `len` and in this case calling `sb.getchars(...)` method is safe. When we consider the explained pair by calling `sb.getchars(...)` after the code in Figure 6.1 is called we will not run the scenario explained in Section 2.3. As a result, we will not be able to find that atomicity bug. Treating each method call to a shared variable as a different location is a future work and when implemented, it will make more precise coverage measurements.

Modifying the Values of Variables on Writes: In the motivating example presented in Section 2.3, the error was due to an atomicity violation. The execution trace which leads to this error is presented in Figure 2.10. Suppose that in this execution trace, thread *t2* calls the `sb.setLength(arg);` with an argument which is equal to the value of the variable `len`. In such a case, even though the pair which creates the bug is covered, the error cannot be detected because the length of the array remains same. As a future work, writes to shared variables should be forced to modify the current values of the variables. If the value of the variable is not changed during the tests, it should be reported to the developer. The developer can create more interesting scenarios by implicitly changing the values of variables at each write to a shared variable. This improvement will increase the chance of detecting a concurrency error in the programs.

BIBLIOGRAPHY

- [1] Eitan Farchi, Yarden Nir, and Shmuel Ur. Concurrent bug patterns and how to test them. In *IPDPS '03: Proceedings of the 17th International Symposium on Parallel and Distributed Processing*, page 286.2, Washington, DC, USA, 2003. IEEE Computer Society.
- [2] Cyrille Artho, Klaus Havelund, and Armin Biere. High-level data races. *Softw. Test., Verif. Reliab.*, 13(4):207–227, 2003.
- [3] Tayfun Elmas, Serdar Tasiran, and Shaz Qadeer. VyrD: verifying concurrent programs by runtime refinement-violation detection. In *PLDI '05: Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*, pages 27–37, New York, NY, USA, 2005. ACM Press.
- [4] M. Erkan Keremoglu, Serdar Tasiran, and Tayfun Elmas. A classification of concurrency bugs in java benchmarks by developer intent. In *PADTAD '06: Proceeding of the 2006 workshop on Parallel and distributed systems: testing and debugging*, pages 23–26, New York, NY, USA, 2006. ACM Press.
- [5] Mayur Naik, Alex Aiken, and John Whaley. Effective static race detection for java. In *PLDI '06: Proceedings of the 2006 ACM SIGPLAN conference on Programming language design and implementation*, pages 308–319, New York, NY, USA, 2006. ACM Press.
- [6] Willem Visser, Corina S. Păsăreanu, and Sarfraz Khurshid. Test input generation with java pathfinder. In *ISSTA '04: Proceedings of the 2004 ACM SIGSOFT international symposium on Software testing and analysis*, pages 97–107, New York, NY, USA, 2004. ACM Press.

- [7] Richard N. Taylor, David L. Levine, and Cheryl D. Kelly. Structural testing of concurrent programs. *IEEE Trans. Softw. Eng.*, 18(3):206–215, 1992.
- [8] M. Factor, E. Farchi, Y. Lichtenstein, and Y. Malka. Testing concurrent programs: a formal evaluation of coverage criteria. In *ICCSSE '96: Proceedings of the 7th Israeli Conference on Computer-Based Systems and Software Engineering*, page 119, Washington, DC, USA, 1996. IEEE Computer Society.
- [9] Orit Edelstein, Eitan Farchi, Yarden Nir, Gil Ratsaby, and Shmuel Ur. Multi-threaded java program test generation. In *JGI '01: Proceedings of the 2001 joint ACM-ISCOPE conference on Java Grande*, page 181, New York, NY, USA, 2001. ACM Press.
- [10] Ken Arnold, James Gosling, and David Holmes. *Java(TM) Programming Language, The (4th Edition)*. Addison-Wesley Professional, 2005.
- [11] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *The Java Language Specification Second Edition*. Addison-Wesley, Boston, Mass., 2000.
- [12] Leslie Lamport. Proving the correctness of multiprocess programs. *IEEE Trans. Software Eng.*, 3(2):125–143, 1977.
- [13] Jeremy Manson, William Pugh, and Sarita V. Adve. The java memory model. In *POPL '05: Proceedings of the 32nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 378–391, New York, NY, USA, 2005. ACM Press.
- [14] Cormac Flanagan and Stephen N Freund. Atomizer: a dynamic atomicity checker for multithreaded programs. *SIGPLAN Not.*, 39(1):256–267, 2004.
- [15] Christoph von Praun and Thomas R. Gross. Static detection of atomicity violations in object-oriented programs. *Journal of Object Technology*, 3(6):103–122, 2004.

- [16] Tayfun Elmas. Runtime Checking of Refinement for Concurrent Software Components. Master's thesis, Electrical And Computer Engineering, Koc University, Istanbul, December 2005.
- [17] Serdar Tasiran and Shaz Qadeer. Runtime refinement checking of concurrent data structures. *Electr. Notes Theor. Comput. Sci.*, 113:163–179, 2005.
- [18] Serdar Tasiran, Tayfun Elmas, Guven Bolukbasi, and M. Erkan Keremoglu. A novel test coverage metric for concurrently-accessed software components. In *FATES*, pages 62–71, 2005.
- [19] Jr. Edmund M. Clarke, Orna Grumberg, and Doron A. Peled. *Model checking*. MIT Press, Cambridge, MA, USA, 1999.
- [20] Tom Henzinger, Ranjit Jhala, Rupak Majumdar, and Gregoire Sutre. Software verification with blast. In *Proceedings of the Tenth International Workshop on Model Checking of Software (SPIN)*, pages 235–239. Lecture Notes in Computer Science 2648, Springer-Verlag, January 2003.
- [21] Thomas A. Henzinger, Ranjit Jhala, Rupak Majumdar, and Grégoire Sutre. Lazy abstraction. *SIGPLAN Not.*, 37(1):58–70, 2002.
- [22] John Hatcliff and Matthew Dwyer. Using the bandera tool set to model-check properties of concurrent Java software. *Lecture Notes in Computer Science*, 2154, 2001.
- [23] Gerard J. Holzmann. The model checker spin. *IEEE Trans. Softw. Eng.*, 23(5):279–295, 1997.
- [24] Thomas Ball and Sriram K. Rajamani. The slam project: debugging system software via static analysis. *SIGPLAN Not.*, 37(1):1–3, 2002.

-
- [25] Willem Visser, Klaus Havelund, Guillaume Brat, Seungjoon Park, and Flavio Lerda. Model checking programs. *Automated Software Engg.*, 10(2):203–232, 2003.
 - [26] Patrice Godefroid. Model checking for programming languages using verisoft. In *Symposium on Principles of Programming Languages*, pages 174–186, 1997.
 - [27] Patrice Godefroid. *Partial-Order Methods for the Verification of Concurrent Systems: An Approach to the State-Explosion Problem*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1996. Foreword By-Pierre Wolper.
 - [28] Hubert Garavel, Radu Mateescu, and Irina Smarandache. Parallel state space construction for model-checking. In *SPIN '01: Proceedings of the 8th international SPIN workshop on Model checking of software*, pages 217–234, New York, NY, USA, 2001. Springer-Verlag New York, Inc.
 - [29] Michael Hind. Pointer analysis: Haven't we solved this problem yet? In *2001 ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering (PASTE'01)*, Snowbird, UT, 2001.
 - [30] Thomas Marlowe, Barbara Ryder, and Michael Burke. Defining flow sensitivity in data flow problems. Technical Report LCSR-TR-249, Rutgers University, July 1995.
 - [31] Radu Rugina and Martin Rinard. Pointer analysis for multithreaded programs. *SIGPLAN Not.*, 34(5):77–90, 1999.
 - [32] Nicholas Sterling. Warlock - a static data race analysis tool. In *USENIX Winter*, pages 97–106, 1993.
 - [33] Cormac Flanagan, K. Rustan M. Leino, Mark Lillibridge, Greg Nelson, James B. Saxe, and Raymie Stata. Extended static checking for java. In *PLDI '02: Proceed-*

- ings of the ACM SIGPLAN 2002 Conference on Programming language design and implementation*, pages 234–245, New York, NY, USA, 2002. ACM Press.
- [34] Gary T. Leavens, K. Rustan M. Leino, Erik Poll, Clyde Ruby, and Bart Jacobs. JML: notations and tools supporting detailed design in Java. In *OOPSLA 2000 Companion, Minneapolis, Minnesota*, pages 105–106, 2000.
- [35] Dawson Engler and Ken Ashcraft. Racerx: effective, static detection of race conditions and deadlocks. In *SOSP '03: Proceedings of the nineteenth ACM symposium on Operating systems principles*, pages 237–252, New York, NY, USA, 2003. ACM Press.
- [36] J. Choi, A. Loginov, and V. Sarkar. Static datarace analysis for multithreaded object-oriented programs, 2001.
- [37] Stefan Savage, Michael Burrows, Greg Nelson, Patrick Sobalvarro, and Thomas Anderson. Eraser: a dynamic data race detector for multithreaded programs. *ACM Trans. Comput. Syst.*, 15(4):391–411, 1997.
- [38] Guang-Ien Cheng, Mingdong Feng, Charles E. Leiserson, Keith H. Randall, and Andrew F. Stark. Detecting data races in cilk programs that use locks. In *SPAA '98: Proceedings of the tenth annual ACM symposium on Parallel algorithms and architectures*, pages 298–309, New York, NY, USA, 1998. ACM Press.
- [39] Rahul Agarwal and Scott D. Stoller. Type inference for parameterized race-free java. In *Proceedings of the Fifth International Conference on Verification, Model Checking and Abstract Interpretation*, volume 2937 of *Lecture Notes in Computer Science*, pages 149–160. Springer-Verlag, January 2004.
- [40] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM*, 21(7):558–565, 1978.

- [41] Serdar Tasiran Tayfun Elmas, Shaz Qadeer. Goldilocks: Efficiently computing the happens-before relation using locksets. Technical Report MSR-TR-2006-163, Microsoft Research, 2006.
- [42] Mark Christiaens and Koenraad De Bosschere. Trade: Data race detection for java. In *ICCS '01: Proceedings of the International Conference on Computational Science-Part II*, pages 761–770, London, UK, 2001. Springer-Verlag.
- [43] Michiel Ronsse and Koen De Bosschere. Replay: a fully integrated practical record/replay system. *ACM Trans. Comput. Syst.*, 17(2):133–152, 1999.
- [44] D. Schonberg. On-the-fly detection of access anomalies. In *PLDI '89: Proceedings of the ACM SIGPLAN 1989 Conference on Programming language design and implementation*, pages 285–297, New York, NY, USA, 1989. ACM Press.
- [45] Eli Pozniansky and Assaf Schuster. Efficient on-the-fly data race detection in multithreaded c++ programs. *SIGPLAN Not.*, 38(10):179–190, 2003.
- [46] Yuan Yu, Tom Rodeheffer, and Wei Chen. Racetrack: Efficient detection of data race conditions via adaptive tracking. In *SOSP '05: Proceedings of the twentieth ACM symposium on Operating systems principles*, pages 221–234, New York, NY, USA, 2005. ACM Press.
- [47] Yaniv Eytani, Klaus Havelund, Scott D. Stoller, and Shmuel Ur. Toward a framework and benchmark for testing tools for multi-threaded programs. *Concurrency and Computation: Practice & Experience*, 19(3):267–279, 2006.
- [48] Scott D. Stoller. Testing concurrent Java programs using randomized scheduling. In *Proc. Second Workshop on Runtime Verification (RV)*, volume 70(4) of *Electronic Notes in Theoretical Computer Science*. Elsevier, July 2002.
- [49] Yosi Ben-Asher, Eitan Farchi, and Yaniv Eytani. Heuristics for finding concurrent bugs. *IPDPS*, 2003.

- [50] Jong-Deok Choi and Harini Srinivasan. Deterministic replay of java multi-threaded applications. In *SPDT '98: Proceedings of the SIGMETRICS symposium on Parallel and distributed tools*, pages 48–59, New York, NY, USA, 1998. ACM Press.
- [51] Klaus Havelund and Grigore Rosu. Monitoring java programs with java pathexplorer. Technical report, 2001.
- [52] Grigore Roşu and Koushik Sen. An instrumentation technique for online analysis of multithreaded programs: Research articles. *Concurr. Comput. : Pract. Exper.*, 19(3):311–325, 2007.
- [53] Jan Tretmans. Testing concurrent systems: A formal approach. In *CONCUR '99: Proceedings of the 10th International Conference on Concurrency Theory*, pages 46–65, London, UK, 1999. Springer-Verlag.
- [54] Hong Zhu, Patrick A. V. Hall, and John H. R. May. Software unit test coverage and adequacy. *ACM Comput. Surv.*, 29(4):366–427, 1997.
- [55] B. Korel. Automated software test data generation. *IEEE Trans. Softw. Eng.*, 16(8):870–879, 1990.
- [56] Cheer-Sun D. Yang, Amie L. Souter, and Lori L. Pollock. All-du-path coverage for parallel programs. In *ISSTA '98: Proceedings of the 1998 ACM SIGSOFT international symposium on Software testing and analysis*, pages 153–162, New York, NY, USA, 1998. ACM Press.
- [57] Arkady Bron, Eitan Farchi, Yonit Magid, Yarden Nir, and Shmuel Ur. Applications of synchronization coverage. In *PPoPP '05: Proceedings of the tenth ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 206–212, New York, NY, USA, 2005. ACM Press.

- [58] Charles Daly, Jane Horgan, James Power, and John Waldron. Platform independent dynamic java virtual machine analysis: the java grande forum benchmark suite. In *JGI '01: Proceedings of the 2001 joint ACM-ISCOPE conference on Java Grande*, pages 106–115, New York, NY, USA, 2001. ACM Press.