박사 학위논문

Ph. D. Dissertation

동시성 커버리지 메트릭을 이용한 멀티쓰레드 프로그램의 효과적이고 효율적인 테스트 생성

Effective and Efficient Test Generation for Multithreaded Programs Using Concurrency Coverage Metrics

홍 신 (洪 申 Hong, Shin)

전산학부

School of Computing

KAIST

2015

동시성 커버리지 메트릭을 이용한 멀티쓰레드 프로그램의 효과적이고 효율적인 테스트 생성

Effective and Efficient Test Generation for Multithreaded Programs Using Concurrency Coverage Metrics

Effective and Efficient Test Generation for Multithreaded Programs Using Concurrency Coverage Metrics

Advisor : Professor Kim, Moonzoo

by

Hong, Shin

School of Computing

KAIST

A thesis submitted to the faculty of KAIST in partial fulfillment of the requirements for the degree of Doctor of Philosophy in the School of Computing . The study was conducted in accordance with Code of Research Ethics[1].

2015. 5. 26.

Approved by

Professor Kim, Moonzoo

[Advisor]

[1]Declaration of Ethical Conduct in Research: I, as a graduate student of KAIST, hereby declare that I have not committed any acts that may damage the credibility of my research. These include, but are not limited to: falsification, thesis written by someone else, distortion of research findings or plagiarism. I affirm that my thesis contains honest conclusions based on my own careful research under the guidance of my thesis advisor.

동시성 커버리지 메트릭을 이용한 멀티쓰레드 프로그램의 효과적이고 효율적인 테스트 생성

홍 신

위 논문은 한국과학기술원 박사학위논문으로
학위논문심사위원회에서 심사 통과하였음.

2015년 5월 26일

심사위원장	김문주	(인)
심사위원	류석영	(인)
심사위원	한태숙	(인)
심사위원	허재혁	(인)
심사위원	Chao Wang	(인)

DCS
20115325

홍 신. Hong, Shin. Effective and Efficient Test Generation for Multithreaded Programs Using Concurrency Coverage Metrics. 동시성 커버리지 메트릭을 이용한 멀티쓰레드 프로그램의 효과적이고 효율적인 테스트 생성. School of Computing . 2015. 123p. Advisor Prof. Kim, Moonzoo. Text in English.

ABSTRACT

To effectively utilize advanced multi-core processors, many software systems today are built as multithreaded programs. One challenge in developing a multithreaded program is to ensure its correctness. Unlike single-threaded programs, the behavior of a multithreaded program depends not only on program input values, but also on thread schedules. To ensure correctness, a multithreaded program must be checked for all thread schedule cases whose numbers are notoriously large due to non-determinism in thread scheduling. Many techniques have been proposed to generate various thread schedules to test multithreaded programs and find concurrency errors. However, none of these techniques can provide effective and efficient fault detection results at the same time scales with respect to the size of a target multithreaded program.

In this dissertation, I present a new technique that utilizes concurrency coverage metrics to generate effective and efficient test executions of multithreaded programs. Concurrency coverage metrics aid in supporting multithreaded program testing by providing reasonable assessments of the testing quality. Unlike structural coverage metrics such as the branch and statement coverage metrics, it has not been well established that concurrency coverage metrics are actually useful for multithreaded program testing, and there exist only a few applications of concurrency coverage metrics for multithreaded program test generation. The first part of this dissertation, through empirical study, attempts to answer the question of whether or not the existing concurrency coverage metrics are actually useful for testing multithreaded programs. Although none of the existing metrics seems to be perfect, the study results show that the existing concurrency coverage metrics are effective at estimating fault detection ability and at providing test generation targets in multithreaded program testing. The second part of this dissertation presents a new concurrency coverage based thread scheduling algorithm that generates test executions to achieve high concurrency coverage fast. In particular, the proposed technique utilizes a new coverage metric called the combinatorial concurrency coverage metric, which supplements the existing concurrency coverage metrics by providing more fine-grained test targets. The experiment results show that the proposed technique generates more effective and more efficient test executions than do the existing multithreaded program test generation techniques. For the last part of this dissertation, I present a concurrency coverage-based regression testing technique for multithreaded programs. The proposed technique utilizes a concurrency coverage metric to identify the changed multithreaded program behavior by a code modification; it then generates the targeted test executions for the identified behaviors to effectively detect regression faults. The experiment results imply that the proposed regression testing technique is more effective and efficient than are the existing techniques at detecting regression faults of multithreaded programs.

Contents

List of Tables

List of Figures

Chapter 1. Introduction

1.1 Challenges in testing multithreaded programs

To effectively utilize advanced multi-core CPUs, many software systems today are developed as *concurrent programs*. Unlike sequential programs that run one stream of operations in their executions, concurrent programs execute multiple streams of operations at the same time. Concurrent programs can inherently deploy multiple processor cores simultaneously to execute multiple streams of operations concurrently, easily achieving high utilization of multi-core processors. Writing concurrent code has become an important programming skill for software developers. A survey of Microsoft software developers presented in 2008 says that 50% of Microsoft product-line developers write concurrent programs to leverage multi-core processors and enhance the computational speed and/or responsiveness of their software products [38].

A *multithreaded program* is one of the most popular types of concurrent programs in practice. A multithreaded program consists of a set of sequential code fragments called *threads* that can run concurrently and interact with each other through shared data structure in their execution. According to an empirical study presented in 2012, 87% of the large-scale open-source C# programs in the study contain multithreaded code [73]. A characteristic of multithreaded programs is *non-deterministic behavior*, which is the result of non-deterministic thread scheduling. The standard multithreaded program semantics allow an arbitrary execution order of operations in concurrently running threads. To fully utilize multi-core processors under different runtime circumstances, most thread schedulers of threading libraries or operating systems exploit this non-determinism for dynamic thread scheduling.

Despite the widespread of multithreaded program development, writing multithreaded programs correctly remains a challenging task for developers because there is no practical verification method to ensure the correctness of the non-deterministic behaviors of multithreaded programs. A multithreaded program may show different results for the same program input (i.e., non-deterministic behavior) because its execution depends not only on program input, but also on non-deterministic execution orders of operations across threads. As a consequence, a multithreaded program can have a *concurrency error*, which is caused by incorrect interactions among concurrent threads. Different from bugs in sequential programs, a concurrency error does not always appear for a specific input value, but only under specific thread schedules. To detect concurrency errors and ensure correctness, a multithreaded program must be checked with all thread schedule cases. However, even for small multithreaded programs, the number of distinguishable thread schedules is enormously large and there is no practical method that can effectively and efficiently verify the correctness of multithreaded programs. For this reason, it is well-known that concurrency errors/bugs are difficult to detect, reproduce, and fix.

The problem of concurrency errors is an actual threat in concurrent program development. According to interviews of skilled software developers [86], the interviewee commonly say that "the hardest bugs to track down are in concurrent code"; most of them agree that "ubiquitous multi-core CPUs are going to force some serious changes in the way software is written", and thus finding concurrency errors and ensuring the correctness of multithreaded programs are some of the most challenging tasks that they face.

To date, testing is the most popular method in practice to find bugs and ensure the correctness of software. To observe various program behaviors and check if any behavior is unintended (i.e., error), testing executes a target program while controlling factors that influence the target program behaviors. Compared with other kinds of verification techniques such as model checking or dynamic/static analyses, testing is more precise as testing checks for actual program behaviors rather than using abstract models. Testing can detect any form of errors, whereas static/dynamic analyses are limited to finding certain types of errors. In addition, with real-world software, testing is more effective than model checking techniques as the state-of-art model checkers are still not scalable for verification of the industrial software. For these reasons, testing is the de facto standard method to ensure the software correctness in practice; many techniques have been developed to generate useful program inputs for the testing of sequential programs.

Unfortunately, conventional testing methods and techniques are not effective for multithreaded programs because multithreaded program behaviors are also influenced by thread schedules at the execution time. A common practice is stress testing which repeats executions of the target programs while providing unusual workload to the thread scheduler in hopes of observing uncommon thread scheduling cases. However, repeating test executions for a certain test environment redundantly generates similar thread schedules and does not effectively increase the likelihood of detecting concurrency errors.

Effective testing of multithreaded programs must generate various thread schedules that induce all behaviors of a target program per program input value; however, generating such effective testing is challenging for the following reasons. First, even for a small-size program, there exists too large a number of distinct thread scheduling cases for this process to be feasible for an ordinary testing budget (i.e., the thread schedule explosion problem). The number of distinguishable thread scheduling choices is exponentially related to the number of operations in an execution (i.e., execution length). For a multithreaded program with T number of threads each of which has N number of operations, the number of possible thread scheduling decisions is $\frac{(T \times N)!}{(N!)^T}$. Another problem of testing all thread schedules is that many distinguishable thread schedules may generate redundant program behavior; thus, efforts to test these thread schedules are useless. Let us consider a multithreaded program with T threads that have N threads each. Suppose that none of these threads commonly accesses to the same shared variable. Although there exist an exponential number of possible thread schedules, all these thread schedules generate the same execution result. A study on concurrency errors shows that the probability of detecting concurrency errors is often very low (> 0.001) without a systematic generation of thread execution order [26]. Therefore, testing of a multithreaded program should be done in such a way as to carefully select thread schedules as test cases so as to obtain effective and efficient concurrency error detection: the selected thread schedules should generate comprehensive program behaviors so as not to miss any errors in the program; however, to save testing cost, each selected thread schedule generates a distinguishable program behavior.

1.2 Limitations of existing testing techniques

For effective and efficient concurrency error detections, conventional test generation techniques for multithreaded programs aim to generate useful thread schedules using certain criteria. In high level, there are three kinds of the conventional test generation techniques: random noise injection-based test generation, bug-directed test generation, and systematic test generation. Figure 1.1 describes the limitations of these three kinds of test generation techniques. These conventional techniques commonly have

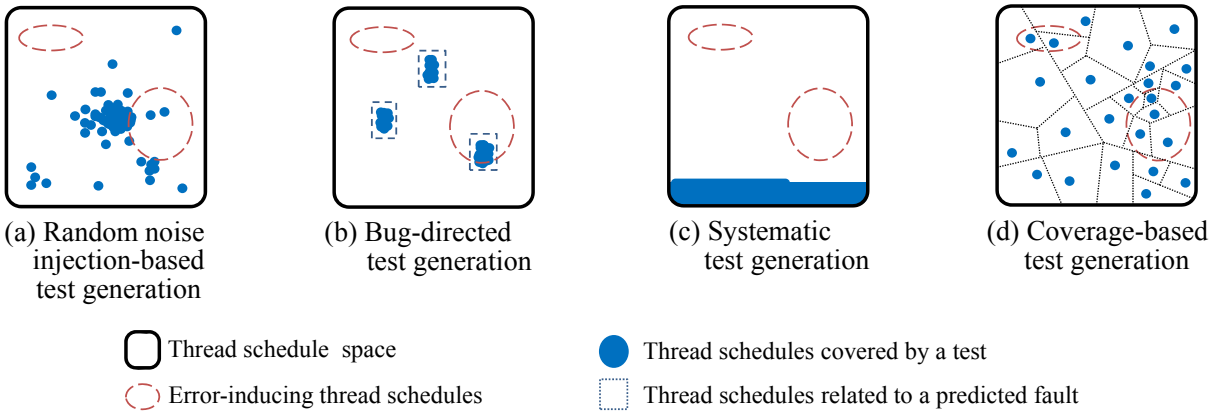


Figure 1.1: Comparison of the conventional test generation techniques with the coverage based test generation technique

limitations for generating highly effective and highly efficient tests at the same time. Discussion of the three kinds of testing techniques is as follows (more discussion is found in Chapter 2):

- Random noise injection-based test generation (Section 2.4.1)
  Random noise injection-based test generation techniques insert random timing delay into a target program, and then execute the modified target programs many times [27, 96]. The intuition of these techniques is that the inserted timing delay diversifies the execution order of the threads to increase the chance of generating error executions. Although these techniques are effective to diversify thread schedule generations, the limitation of these techniques is that there is no guarantee that random noise injection-based test generation techniques can keep generating unseen program behavior, as shown in Figure 1.1(a). For this reason, these techniques are not effective at detecting concurrency errors with few error-inducing thread schedules [75].

- Bug-driven test generation (Section 2.4.2)
  Bug-driven test generation techniques use static/dynamic analyses to predict possible concurrency errors before testing, and then these techniques generate the thread schedules targeted to induce each predicted concurrency error [48, 75, 88]. As can be seen in Figure 1.1(b), these techniques generate thread schedules toward predefined targets. Although these techniques are effective and efficient at detecting certain types of concurrency faults, bug-directed techniques cannot be used to detect arbitrary concurrency errors that do not match any predefined pattern.

- Systematic test generation (Section 2.4.3)
  Systematic test generation techniques (or model checking techniques) rigorously explore all thread schedule decisions for a given input value [105, 109]. As these techniques can explore all possible thread schedules, they can detect all existing concurrency faults in a multithreaded program if a sufficient amount of testing time is given. However, can be seen in Figure 1.1(a), with a limited testing time, these techniques may generate a small subset of thread schedules that do not induce concurrency errors because there exist tremendous number of thread schedules. To compensate this limitation, several techniques utilize heuristic search strategies to resolve the low efficiency problem; however, currently existing techniques do not show sufficient performance to effectively check large size multithreaded programs.

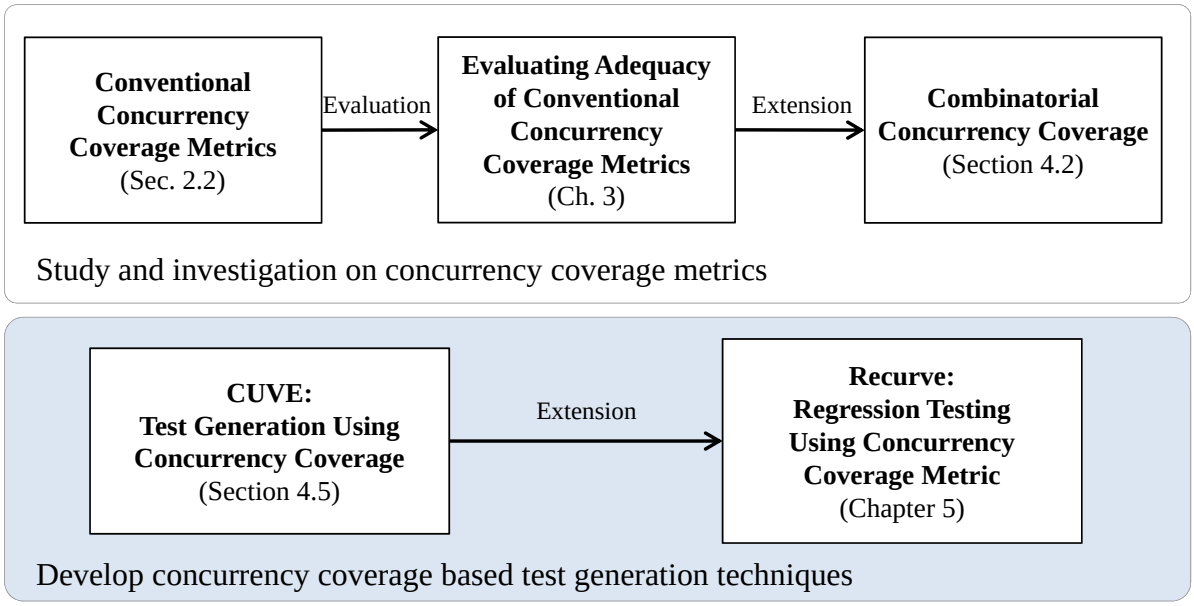


Figure 1.2: Overview of this dissertation

Beside testing techniques, another approach for effectively detecting concurrency faults is to predict suspicious instructions/operations that may induce concurrency errors via static/dynamic analyses. These techniques can find concurrency errors without generating actual error executions. However, these techniques commonly have limited fault detection ability because their analyses are able to find out only specific types of concurrency faults. Furthermore, due to the limited precision of the analyses, predicted concurrency faults may not result in any actual error (i.e., false positives).

1.3 Approach: generating tests to achieve high concurrency coverage

In this dissertation, I present a new approach of multithreaded program test generation utilizing concurrency coverage metrics (see Figure 1.2). To assess the quality of the multithreaded program testing (i.e., comprehensiveness and necessity), various concurrency coverage metrics are proposed. A concurrency coverage metric derives a set of test requirements from a target program code. Similar to branch/statement coverage metrics for the testing of sequential programs, concurrency coverage metrics generate a test requirement to check whether or not a certain thread interaction involved with specific code entities is executed at least one execution in a test. A set of test requirements for a concurrency coverage metric can be used to measure the expected effectiveness and also to identify unexamined program behaviors of a test.

I expect that a test generated toward achieving high concurrency coverage will be effective and efficient at detecting concurrency errors because the concurrency coverage metrics provide useful targets corresponding to various thread interaction cases of a target program. Figure 1.1(d) illustrates the coverage based test generation technique. As concurrency coverage metrics provide a moderate number of comprehensive test generation targets, I expect that the coverage based test generation technique can effectively and quickly detect arbitrary concurrency errors.

There is currently active research on test case generation techniques that aim to achieve high branch/statement coverage because a strong positive correlation of coverage achievement and fault detection ability has been empirically shown. However, there has been little research on how concurrency coverage metrics can be utilized for effective test case generation for multithreaded programs. To develop effective testing methods using concurrency coverage metrics, I initially explored the following questions:

- *Do concurrency coverage metrics properly measure the effectiveness of tests of a multithreaded program?*

  Unlike branch/statement coverage metrics for sequential program testing, there was no rigorous study that examined the impact of concurrency coverage metrics on the effectiveness of multithreaded program testing. To answer this question, I conducted empirical studies to check whether or not a conventional concurrency coverage metric would be a good predictor of testing effectiveness, and whether or not the set of test requirements by the existing concurrency coverage metric would be proper test generation targets.

- *How can we generate test cases that achieve high concurrency coverage fast?*

  I developed a new thread scheduling technique that dynamically guides a running execution to achieve high concurrency coverage. Although there exist techniques that has mechanisms to increase the possibility of achieving more coverage, they have limitations for achieving high effectiveness and high efficiency at the same time.

To generate effective tests, I investigate the conventional concurrency coverage metrics and propose new test generation techniques that utilize concurrency coverage metrics for better test generations. More specifically, the results of my study and the techniques I develop can be summarized in the following three points:

- I empirically evaluate whether or not coverage increase in a coverage metric properly estimates the increase in testing effectiveness and whether or not a testing designed to achieve maximum coverage shows higher testing effectiveness than another testing of the same size. To quantify these things, I used mutation testing for concurrent programs. I found that there are certain existing concurrency coverage metrics that provide good estimations of the effectiveness of tests, whereas the other coverage metrics do not properly estimate the test effectiveness. In addition, I found empirical evidence that no existing concurrency coverage metric is perfect.

- I develop a coverage-based thread schedule generation technique to achieve high concurrency coverage fast. The technique is based on two key ideas. First, the technique utilizes a dynamic analysis technique that precisely estimate likely feasible test requirements. Second, the technique utilizes a new concurrency coverage metric called the combinatorial concurrency coverage metric which generate more useful sets of test requirements than the existing concurrency coverage metrics. I implement the testing technique with the proposed thread schedule generation algorithm for multithreaded Java programs. The experiment shows that my technique generates test cases that achieve higher coverage faster than other random testing techniques do.

- I define a new concurrency coverage metric *combinatorial concurrency coverage metrics* and extend the thread schedule generation technique that achieves high coverage fast for a target multithreaded program. The combinatorial concurrency coverage metric can capture more diverse thread schedules than the conventional concurrency coverage metrics. Through a series of experiments with

multiple faulty programs including real-world bugs, I demonstrate that my technique is more effective and efficient in concurrency fault detections than other techniques.

- I extend the coverage-based thread scheduling algorithm to generate regression testing of multithreaded programs. The proposed technique utilizes a concurrency coverage metric to detect and target the changed behavior of a target program for a program modification. The experiment result shows that the proposed regression technique is more effective and efficient at detecting regression faults than are the existing techniques.

1.4 Structure of the dissertation

The remainder of the dissertation is structured as follows. First, in Chapter 2, I present the background and related work for multithreaded program testing and analysis techniques . Especially, I present a classification of race bugs and detection techniques for these bugs because a lot of previous research efforts on multithreaded program analyses have been concentrated on detecting race bugs. Chapter 3 presents the empirical evaluation of the presented concurrency coverage metrics on testing effectiveness. I report experiment results for measures of correlations of the coverage achievement and the fault detection ability in test suites and also the fault detection ability of the test suites generated for maximum coverage achievement with minimum size. In Chapter 4, I present a new test generation technique for multithreaded programs based on concurrency coverage metrics. For effective concurrency fault detection, I propose a new concurrency coverage metric called *combinatorial concurrency coverage*; I also propose thread scheduling algorithms that utilize the conventional and the new coverage metrics to achieving high fault detection fast. The experiment results indicate that the proposed test generation technique shows better performance in fault detection and also in coverage achievement than do other multithreaded program testing techniques. Finally, in Chapter 5, I present an extension of the coverage-based test generation technique for regression testing of multithreaded programs. I also present the experiment results, which show that the proposed regression technique is effective and efficient at detecting regression faults in multithreaded programs. I conclude this dissertation in Chapter 6 with a consideration of future work.

Chapter 2. Background and Related Work

In this chapter, I present the background on multithreaded program testing including concurrency bugs, concurrency coverage metrics, and the existing testing techniques for multithreaded programs. [1] In Section 2.1, I explain the basics of multithreaded programs and concurrency bugs. In Section 2.2, I present the existing concurrency coverage metrics proposed in earlier work. As a part of the related work, I present a survey on various race bugs and their detection techniques for multithreaded programs in Section 2.3, because the largest portion of the research effort on the multithreaded program analysis has been concentrated on detecting/predicting race bugs Last, I discuss the existing test generation techniques for multithreaded programs in Section 2.4.

2.1 Concurrency bugs in multithreaded programs

2.1.1 Multithreaded programs and executions

A multithreaded program consists of a finite set of threads, thread local variables, and shared variables. Each thread is a sequential program that manipulates its local variables, manipulates the shared variables which can be manipulated by any thread in the same multithreaded program, and synchronizes with other threads to control execution orders in different threads. A thread interacts with other concurrently running threads by executing synchronization operations and by updating shared variables.

An execution of multithreaded program is a set of thread executions each of which is a finite sequence of operations. A thread accepts given input data, and then reads and writes local and shared variables throughout an execution. Unlike sequential program executions, the multiple threads in a multithreaded program executes concurrently either in a parallel manner, or an interleaving manner. With multiple parallel processors, concurrent threads are executed by different processors simultaneously; otherwise, a single processor interleaves executions of multiple concurrent threads with time sharing (i.e., interleaved executions).

In a multithreaded program execution, basically there is no ordering constraint on two operations that are executed in two different threads at the same time. Rather, their execution order in an execution is decided arbitrary in runtime. Thus, different executions of a same multithreaded program with a same input data may have different execution order of concurrent operations.

Synchronization refers to execution ordering constraints imposed by programmers. Since multiple threads can manipulate a same shared variable concurrently, the result of a multithreaded program execution depends on in which order threads write/read the shared variables. To control concurrency and guard the integrity of shared variables, programmers write synchronization instructions (e.g., binary lock, conditional variable) in a multithreaded program to enforce certain execution ordering constraints such as mutually exclusive execution of certain code fragments, strict ordering between two code fragments.

A multithreaded program is executed with input data and with a thread schedule. As similar to sequential programs, the input data defines the starting state of an execution. A thread schedule is

[1] A part of this chapter was published in the STVR journal [42]

the execution orders of operations in different threads, which is decided by the thread scheduler of an execution environment. When a multithreaded program runs, a thread scheduler either in an operating system or in a threading library decides which processor executes which thread at a time. Therefore, the decisions made by a thread scheduler throughout an execution results in the execution order of operations in different threads.

A thread schedule of an execution corresponds to the execution orders of operations among threads in the execution. An interleaved execution where the operations from all threads are totally ordered is a model that represents a thread schedule of a multithreaded program execution. With the sequential consistent (strong) memory model, interleaved executions can represent all possible thread schedules. An interleaved execution is also a sound model of a multithreaded program execution with relaxed (weak) memory models where operations in different threads are partially ordered. [2]

A thread schedule of a multithreaded program is generally non-deterministic. This means that a thread scheduler for a multithreaded program is possible to generate any thread schedule that is feasible. Even with a same input data, a multithreaded program can be executed with various thread schedules because there is no restriction of thread schedules except for the synchronization constraints induced by the synchronization instructions in the multithreaded program.

2.1.2 Concurrency errors and concurrency bugs

A multithreaded program has a concurrency error if, depending on thread schedules, the multithreaded program execution fails (i.e. crashes, violates invariants, or generates wrong output). An error is not a concurrency error when the multithreaded program always generates failing executions for a certain input data regarding regardless of thread schedules. Concurrency errors occur when the programmer misuses synchronization instructions which allow unintended interferences in concurrent thread executions.

A concurrency bug (concurrency fault) refers to a set of instructions that causes a concurrency error of a multithreaded program. In theory, it is not trivial to define the concurrency bug correspond to a concurrency error because the error might be caused by complex interactions of many instructions. However, many studies on the concurrency errors in practice show that there are common concurrency bug patterns, for instance, data race bugs and atomicity violations. For this reason, there have been many research works that aim to detect known concurrency bug patterns and propose new concurrency bug patterns. A concurrency bug detection technique such as data race detectors detects/predicts the instructions that are matched with the pattern of a certain kind of concurrency errors.

In the previous works, concurrency bugs are categorized as the deadlock bug and the race bug [3], and there have been many bug patterns presented for each category. A deadlock bug induces an indefinite waiting of a set of threads each of which is blocked by the other threads under certain thread schedules. A deadlock bug is harmful because the bug can make the running program fails to complete its execution. For instance, the cycle locking pattern, the most popular pattern of the deadlock bugs, defines a set of nested lock instructions that can induce deadlock errors depending on their lock acquisition orders.

A race bug leads an execution to violate concurrency requirements intended by programmers which

[2] The interleaved execution model does not completely represent all executions with the weak memory model because an execution with the weak memory model also depends on non-deterministic internal states of multi-core processors.

[3] Livelock is another kind of concurrency bugs which violates liveness properties in reactive systems. However, I do not discuss the livelock bugs because I assume that each thread execution is finite (i.e., target programs are not reactive systems).

result in invalid states or invalid execution results. For example, the data race bug detection techniques determine shared variable accesses without a consistent locking as race bugs because the concurrent execution of these accesses may result in unexpected states of the shared variables. As a lot of research efforts have been concentrated on developing concurrency bug detection techniques for various types of race bugs, I will discuss more on different types of race bugs and their detection techniques in a later section 2.3.

2.2 Concurrency coverage metrics

2.2.1 Overview

A test coverage metric derives test requirements from a target program artifact. A code coverage metric refers to a test coverage metric that generates test requirements from a target program code. A test requirement is a predicate over a program execution. A test (or a test suite) satisfies (or covers) a test requirement when there is at least one execution that satisfies the test requirement in the test. The coverage level (or simply, coverage) of a test with respect to a coverage metric is the ratio of covered test requirements by the test to the number of all generated test requirements by the coverage metric. In general, a test satisfies a coverage metric when the test satisfies all test requirements by the coverage metric for a target program. When a testing uses a coverage metric , a testing process is obligated to have a test case to meet every generated test requirement.

A structural coverage metric generates test requirements to check whether every specific program construct is ever executed (e.g., statement coverage), or a specific condition of a conditional branch is every satisfied (e.g., branch coverage, MC/DC coverage) in single threaded program testing.

A concurrency coverage metric is a code coverage metric specialized to generate test requirements from multithreaded program code. Concurrency coverage metrics aim to generate a set of test requirements that check various thread interaction cases for multithreaded program test executions. Although the structural coverage metrics generate meaningful test requirements for effective single-threaded program testing, these are not able to check presence of certain thread interactions, thus, these are not effective to be used for a multithreaded program testing purpose, where a test should repeat a program execution with a same input value while inducing different thread scheduling.

Various concurrency coverage metrics have been proposed by researchers. These concurrency coverage metrics derive test requirements from synchronization constructs or data access constructs that may manipulate shared variables in a target program. Each concurrency coverage metric is based on an intuition to capture important concurrent execution aspects whose impacts may or may not contribute to concurrency errors in a multithreaded program. Thus, two test requirements derived from a same program construct by two different coverage metrics may have different satisfaction conditions and require a test for different executions to cover.

In this section, I select the eight representative concurrency coverage metrics as shown in Table 2.1, and describe them in full detail. To make the selection concise while having diversity in the collection, I first classify concurrency coverage metrics with the following two aspects–the number of program constructs that define a test requirement, and the construct type that defines a test requirement and then I select at least one coverage metrics that have unique definitions for each coverage metrics type. For the number of program constructs, concurrency coverage metrics can be categorized as singular (i.e., each test requirement is associated with one program construct) or pairwise (i.e., each test requirement

Table 2.1: Overview of eight concurrency coverage metrics

	Synchronization operation	Data access operation
Singular	Blocking [27], Blocked [27]	LR-Def [56]
Pairwise	Blocked-Pair [102], Follows [102], Sync-Pair [41]	PSet [119], Def-Use [98]

is associated with a pair of program constructs). For the other aspect, the concurrency coverage metrics can be categorized as one defined for synchronization program constructs and data accesses program constructs. The selection covers the coverage metrics each of which cannot be defined as a combination of other coverage metrics (i.e., primitive).

The selection contains the thee singular concurrency coverage metrics LR-Def, Blocked, and Blocking and the five pairwise concurrency coverage metrics. LR-Def is a singular data access-based concurrency coverage criterion that generates test requirements for write operations in a target program. Blocked and Blocking are singular synchronization-based coverage metrics that derive test requirements for every lock operation or synchronized block in a target program. For the pairwise data access-based coverage metrics, I select Def-Use and PSet. Note that PSet is almost equivalent to access-pair and location-pair. Thus, I only select PSet as a representative of these. Sync-Pair, Follows, and Blocked-pair are pairwise synchronization-based coverage metrics. Both Sync-Pair and Follows are to check consecutive lock ordering between two lock operations, but have slightly different definitions. Blocked-pair is to check whether or not a lock operation makes the other lock operation blocked.

2.2.2 Concurrency coverage metric definition

In this section, I describes the definition of the test requirements from the eight concurrency coverage metrics. The formal definition of these test requirements are presented in Appendices A.

Synchronization based coverage

Blocked. The *Blocked* coverage criterion [27] defines a test requirement *Blocked(l)* for a synchronization statement located at a program location l. Briefly speaking, a Blocked test requirement *Blocked(l)* is satisfied by an execution if one execution of the statement at l is blocked for acquiring a lock by other threads.

Blocked-Pair. The *Blocked-Pair* coverage criterion defines a test requirement *Blocked-Pair*(l_1, l_2) for two synchronization statements located at a program location l_1 and another location l_2, correspondingly. An execution satisfies *Blocked-Pair*(l_1, l_2) when there are one `lock-hold` action for l_1 and another `lock-acquire` action for l_2 blocked by the locking effect of the `lock-hold` action.

The Blocked-Pair coverage criterion is an extension of the Blocked coverage criterion and the Blocking coverage criterion. An execution that satisfies *Blocked-Pair*(l_1, l_2) also satisfies *Blocking*(l_1) and *Blocked*(l_2).

Blocking. The *Blocking* coverage criterion defines a test requirement *Blocking(l)* for a synchronization statement located at a program location l. A Blocking test requirement *Blocking(l)* is satisfied by an execution if there exist a `lock-hold` action for l that holds a lock variable and a following `lock-acquire` action on the same lock variable that is blocked due to the effect of the `lock-hold` action. A Blocking test

requirement is the counter-part of the Blocked test requirement for a same synchronization statement.

Follows. The *Follows* coverage criterion defines a test requirement for two synchronization statements. A Follows test requirement for two synchronization statements l_1 and l_2 is satisfied when two threads hold a same lock consecutively by the two statements l_1 and l_2 in sequence.

Sync-Pair. The *Sync-Pair* coverage criterion generates test requirements defined over two synchronization statements. A Sync-Pair test requirement *Sync-Pair*(l_1, l_2) checks whether two threads hold a same lock consecutively by two synchronization statements at l_1 and l_2 in order. The Sync-Pair coverage criterion is similar to Follows. The difference is that Sync-Pair generates a test requirement for two consecutive `lock-hold` actions executed by a same thread whereas Follows does not.

Data access based coverage

Def-Use. The Def-Use coverage criterion generates a test requirement for a pair of data write statement and data read statement and a pair of a data write statement and another data write statement. A Def-Use test requirement is satisfied by an execution when the former data write statement defines a value of a data variable and then the latter statement manipulates the data variable while there is no other data write statement defines the same variable between the two statements in the execution. The Def-Use coverage criterion is the same as the def-use (DU) coverage criterion used for the single-threaded program testing.

LR-Def. The LR-Def coverage criterion generates two test requirements for a data read statement: a L-Def test requirement and a R-Def test requirement. The L-Def test requirement for a data read statement l is satisfied by an execution if there is a data write action executed by the same thread that defines the value of a data variable that the read statement reads. The R-Def test requirement for a data read statement l is satisfied by an execution if the data read statement reads the value of a data variable defined by a data write statement executed by a different thread than the data read statement.

PSet. The PSet coverage criterion generates a test requirement for two data access statements that manipulates a same variable and at least one of the two is the write statement. More specifically, A PSet test requirement is defined for a pair of a read statement and a write statement, a pair of a write statement, and a read statement, or a pair of two write statements. PSet aims to check all immediate data dependencies in two data access statements executed by two different threads.

2.2.3 Assessing effectiveness of concurrency coverage metrics

In work on concurrency coverage metrics, the effectiveness of achieving high coverage has been argued for primarily through analytical comparisons between coverage definitions and concurrency fault pattern, such as those involving data races and atomicity violations [56, 102, 109].

Trainin *et al.* [102] note that concurrency faults are related to certain test requirements for the *Blocked-Pair* and *Follows* concurrency coverage metrics, which suggests that achieving high levels of coverage should correlate with testing effectiveness. Wang *et al.* [109] highlight how data races or atomicity violations may be triggered by satisfying *HaPSet* test requirements. Neither analysis empirically demonstrates the benefits of achieving higher coverage.

Tasiran *et al.* [98] evaluate the *location-pair* metric empirically, and compare it to two other coverage metrics (*method-pair* and *def-use*) with respect to the correlation between coverage and fault detection. The study uses two case examples and generates faulty versions via concurrency mutation operators and manual fault seeding. However, the scope of this study is limited for the three metrics and the two case examples.

2.3 Survey on race bugs and the detection techniques

Developing bug-free multithreaded applications is challenging due to non-deterministic thread scheduling, which generates a large number of diverse program behaviors to be checked for correctness. To make multiple threads behave correctly, multithreaded applications enforce synchronization among the threads. A mistake in synchronization, however, can cause *race bugs*, which make a target program execute abnormally with unintended accesses to shared resources. To address this problem, many race bug detection techniques have been developed with predefined *race bug patterns*. These techniques check if execution trace or program code matches the patterns through static or dynamic analyses to identify race bugs.

Despite the wealth of literature on race bug detection techniques, it is difficult to analyze and compare them due to their use of various race bug patterns. This is because different race bug detection techniques define their race bug patterns using different methods, such as the program analysis technique (e.g., dynamic analysis, static analysis, etc.), views on the program execution (e.g., totally ordered execution, partially ordered execution, etc.), target synchronization primitives (e.g., binary lock, wait/notify, etc.), and target memory models (weak memory model, sequentially consistent memory model, etc.). Furthermore, these techniques define the target race bug patterns using different terminologies, which makes the description of the race bug detection techniques difficult to understand.

In this section, as the first step to alleviate the aforementioned problems, I define a formal *execution model* which provides an abstract view on the program execution (Section 2.3.2). By using the proposed *single* execution model, I can define various target race bug patterns from different race bug detection techniques. Then, I compare and classify the techniques according to their target race bugs. I classify race bugs depending on whether or not a bug violates *operation block* specification, and also depending on whether or not the bug violates *data association* specification (Section 2.3.1). Figure 2.1 shows the relationships among the following four classes of race bugs:

- *Data race bug*: operations that manipulate a shared variable in different threads without synchronization (Section 2.3.3)
- *Block race bugs*: operations that violate an operation block specification (Section 2.3.4)
- *Multi-data race bugs*: operations that violate a data association specification (Section 2.3.5)
- *Multi-data block race bugs*: operations that violate both operation block specifications and data association specifications (Section 2.3.6)

I classify race bug detection techniques that do not utilize operation block nor data association specification as *data race bug* detection techniques (Section 2.3.3). Techniques that use the lockset algorithm [84] belong to this class. I classify race bug detection techniques that utilize operation block specification, but not data association specification as *block race bug* detection techniques (Section 2.3.4). This class includes atomicity violation detectors and serializability violation detectors. On the other hand, *multi-data race bug* detection techniques utilize data association specification, but not operation block specification (Section 2.3.5). *Multi-data block race* bug detection techniques utilize both operation block and data association specifications (Section 2.3.6).

I have classified 43 race bug detection techniques in terms of the above four classes: data race bug detection techniques, block race bug detection techniques, multi-data race bug detection techniques, and multi-data block race bug detection techniques. In addition, I have compared the key mechanisms used by the race bug detection techniques to identify the target race bug patterns in program execution or

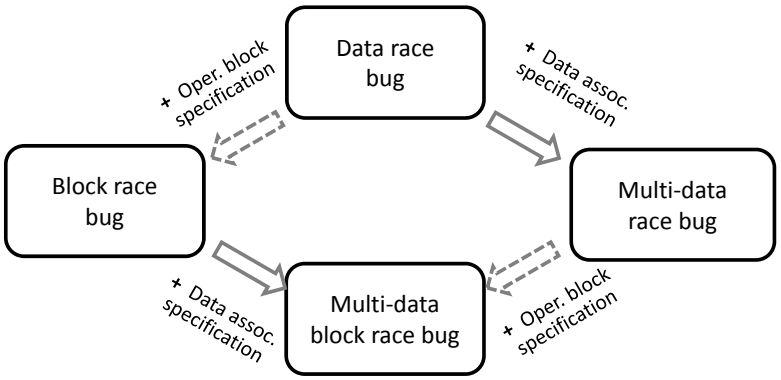


Figure 2.1: Relationship among the four classes of race bug detection techniques

code (see Tables 2.2-2.5). Thus, this survey can help researchers obtain a clear top-down view of various race bug detection techniques for multithreaded programs.

For example, suppose that I would like to know if Velodrome [35] can detect the race bugs that AVIO [59] cannot detect. The answer to the question is not simple because these techniques define race bugs using different notions of correctness, namely, conflict serializability in Velodrome and access interleaving invariants in AVIO. In addition, Velodrome is designed for Java programs with `synchronized` blocks as target synchronization primitives, whereas AVIO analyzes C programs with `lock()`/`unlock()` as target synchronization primitives. Thus, it is difficult to compare the race bug detection capabilities of these two techniques. In contrast, our classification shows that AVIO is a *block race bug* detection technique while Velodrome is a *multi-data block race bug* detection technique. This information allows us to simply answer "yes" to the question above since the block race bug class is a strict subset of the multi-data block race bug class (Section 2.3.7). As another example, from Table 2.3, I know that PENELOPE [94] aims to detect the same class of race bugs targeted by AVIO. However, PENELOPE detects more race conditions as race bugs than AVIO since PENELOPE considers two write operations of different threads as a conflict while AVIO does not (see the fourth column of Table 2.3). In addition, PENELOPE takes the effect of lock synchronization into account, whereas AVIO does not (see the last column of Table 2.3). Thus, PENELOPE and AVIO might detect different block race bugs.

The contributions of this survey are as follows:

- The classification of the 43 techniques provides a systematic overview on race bug detection techniques, which helps researchers understand and improve race bug detection techniques.
- The systematic characterization and concrete examples of a broad range of race bugs can help field engineers avoid race bugs in their multithreaded programs.

2.3.1 Overview of survey

A *race bug* refers to a multithreaded program fault that causes race conditions, which in turn make the program execute incorrectly (i.e., reaching an invalid state). A multithreaded program has a *race condition* if an execution order of two operations is not fixed (i.e., non-deterministic) and their execution result varies depending on the order. [4] Race bug detection techniques detect race bugs by checking if

[4] Race conditions do *not* necessarily cause incorrect program executions. Programmers may utilize race conditions for better processor utilization and responsiveness to requests, since race conditions enable various thread execution orders.

the target program can violate synchronization requirements, which must be satisfied to control multiple threads correctly. Race bug detection techniques typically work in the following three steps:

1. modeling concurrent program behaviors and synchronization requirements;
2. identifying operations that are involved with *race conditions* ;
3. checking if the identified operations can violate the synchronization requirements.

In this paper, I classify race bug detection techniques according to the usage of *operation block* specifications and *data association* specifications, which are orthogonal synchronization requirements for correct multithreaded executions.

- *Operation block* (S_{OB} of Definition 1 in Section 2.3.2)
  An operation block specification is defined as a sequence of consecutive operations of a thread, which are intended to execute without interference from other threads. A program is considered to *satisfy* an operation block specification if a specified operation block always results in a state (regardless of interference from other threads) that is equivalent to the resulting state of the non-interfered operation block.

  Race bug detection techniques receive an atomic region specification that specifies an operation block in the target program. [5] Some race bug detection techniques provide annotation mechanisms for users to specify atomic regions on the target program code [24, 35, 39, 94, 116, 123]. Other techniques automatically infer an operation block specification by using predefined code patterns or execution patterns [17,19,53,59–61,97,112]. Still other techniques simply consider executions of function/method body of certain types as operation blocks [5, 6, 106, 111].

  An operation block corresponds to an execution of "atomic code block" [35], "unit of work" [39], or "transaction" [111] that occur in the literature. Instead of using these terms, I decided to use a new term "operation block", which delivers the high-level meaning clearly because the existing terms defined in specific contexts are not suitable for explaining different techniques.

- *Data association* (S_{DA} of Definition 1 in Section 2.3.2)

  Variables of a data structure are often correlated to each other and may have implicit or explicit invariants. Thus, concurrent accesses to such variables should be synchronized and coordinated to satisfy the invariants/relations. Some race bug detection techniques receive the relation of correlated variables as a requirement specification, and check the "correctness" of access operations to the variables. In other words, they check if these operations are coordinated correctly to satisfy the invariants/relations over correlated variables.

  Data association specifications are obtained from the structure of composite data types (e.g. `struct`, object, array) [79, 107], or obtained from user-given specifications [39, 60, 97]. Some techniques implicitly infer data association specifications from access patterns to variables [5, 35, 57, 106, 116]. Data association corresponds to the terms "multi-variable correlation" [57], "atomic-set" [39], "causal dependency between two variables" [24], or "view" [5] that occur in the literature.

I classify race bugs into four classes depending on the usage of operation block specifications and/or data association specifications in the race bug definition; race bug detection techniques are then classified

[5] Atomic region specification should not be confused with atomic constructs such as `synchronized` blocks in Java.

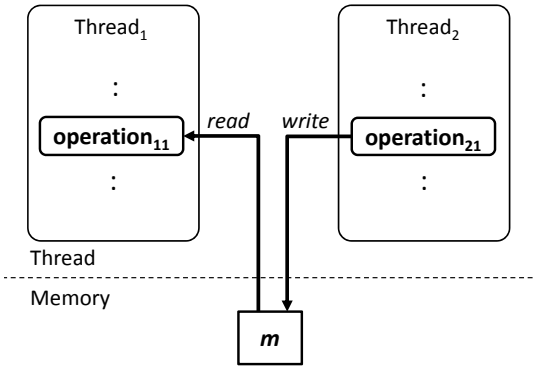


Figure 2.2: A data race bug

according to their target race bug class. The high-level overviews of the four race bug classes are as follows (the formal race bug definitions are described in Sections 2.3.3–2.3.6).

- *Data race bug* (Section 2.3.3)
  A data race bug is defined as two operations of different threads that concurrently read and write (or both write) data on the same memory location without synchronization. Figure 2.2 illustrates an example of a data race bug: operation_{11} of Thread_1 and operation_{21} of Thread_2 can update the same memory location m without synchronization. A data race bug coincides with "data race" in Netzer *et al.* [70].

- *Block race bug* (Section 2.3.4)
  A block race bug is defined as three operations p, p', and q, where p and p' of the same thread belong to an operation block, and q of another thread interferes the intended data-flow between p and p' (this intention is specified by an operation block). A block race bug can be harmful because this interfered data-flow may result in unintended program behaviors and raise an error. Figure 2.3 illustrates an example of a block race bug; operation_{21} of Thread_2 interferes the intended data-flow on m between operation_{11} to operation_{12} of Thread_1. Block race bug is called as "AI (access interleaving)-invariant violation" [59], or "atomicity violation" [53, 61, 76] in the literature. [6]

- *Multi-data race bug* (Section 2.3.5)
  A multi-data race bug is defined as two operations of two different threads that manipulate data associated memory locations without proper synchronization. The multi-data race bug definition extends the data race bug definition to check conditions on data association relations. Figure 2.4 shows an example of a multi-data race bug. Since the accesses to the two data associated memory locations (e.g., m_1 and m_2) are not synchronized, the data structure may become inconsistent. The class of multi-data race bugs covers "multi-variable race" [57] and "object race" [107] in the literature.

- *Multi-data block race bug* (Section 2.3.6)
  A multi-data block race bug is defined as three operations such that two of these are executed by a thread in the same operation block, the other operation is executed by another thread, and the three operations manipulate memory locations with data association relations. The multi-data block

[6] Some papers use the term "atomicity violation" to refer to multi-data race bug (e.g., [35]).

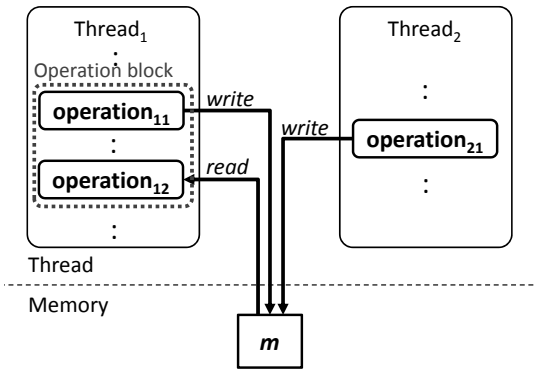


Figure 2.3: A block race bug

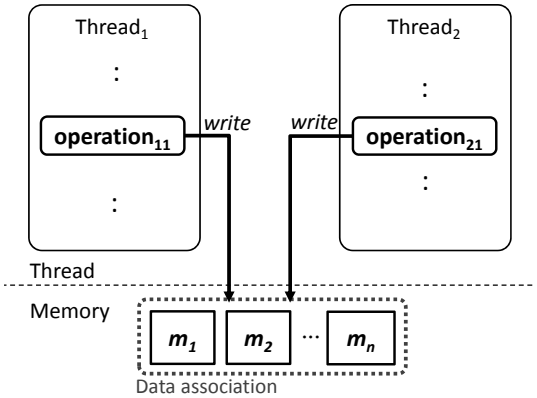


Figure 2.4: A multi-data race bug

race bug definition extends the block race bug definition with an additional data association check. Figure 2.5 describes an example of a multi-data block race bug. As a result of interleaved executions of the three operations, the execution may have unintended combinations of data-flows on the data associated variables. The multi-data block race bug definition corresponds to "high level data race" [5], "atomic-set serializability violation" [39, 104], "multi-variable atomicity violation" [57], or "atomicity violation" [35] in the literature.

2.3.2 Execution model of multithreaded programs

In this section, I formally define an execution model to represent the program executions.

Overview

A race bug detection technique constructs and analyzes an execution model that represents the concurrent executions of a target multithreaded program. Dynamic race bug detection techniques execute the target program to construct an execution model of the program based on the execution traces. Static techniques construct a predictive execution model from the target program code by analyzing the control flow, data flow, and synchronization effects of the program. Most race bug detection techniques have their own definitions of execution models. To describe various definitions of the execution models used

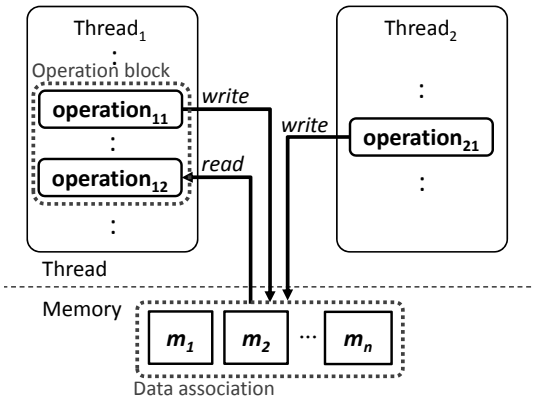


Figure 2.5: A multi-data block race bug

```
  void fun1() {          void fun2() {
1: x = 1;              11: synchronized(m){
2: synchronized(m){    12:  y = 0 ;
3:  y = 1;             13:  x = 0 ;
4: }                   14: }
  }                       }
```

(a) Program code

```
    -th1:fun1()-        -th2:fun2()-
1:  x = 1
2:  lock(m)
3:  y = 1
4:  unlock(m)
                    11:  lock(m)
                    12:  y = 0
                    13:  x = 0
                    14:  unlock(m)
```

(b) Observed execution case

Figure 2.6: Example of a target program code and an observed execution

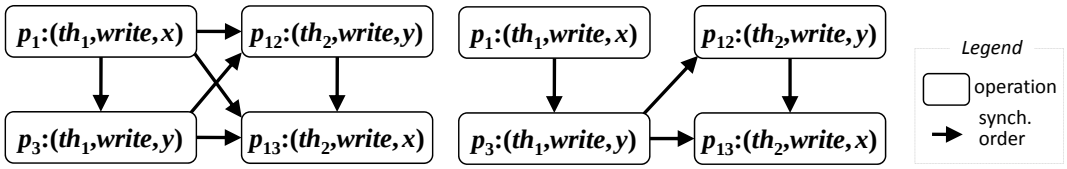
by race bug detection techniques, I define a general execution model that captures the essential aspects of concurrent executions without language- or analysis-technique-specific features.

Our execution model represents the executions of a multithreaded program as *operations* and *synchronized order relations* of the operations. In the model, an operation represents an atomic action of the target program execution (an operation is either read or write). A synchronized order relation represents the order of operations enforced by synchronization. In addition, our execution model represents *operation blocks* as a relation on operations and *data associations* as a relation on the variables of a target program. These specifications are used to define the block race bugs, the multi-data race bugs, and the multi-data block race bugs.

Examples of Execution Models.

This section shows how two different race bug detection techniques can be compared clearly by using our execution model. Suppose that I have a target program (Figure 2.6(a)) and an observed execution of the program (Figure 2.6(b)). Suppose that I would like to compare FastTrack [33] and Eraser [84] both of which target data race bugs. To check if two operations can execute concurrently without synchronization, FastTrack utilizes *happens-before* relation and Eraser utilizes *lockset* algorithm.

Using our execution model definition, I show that FastTrack and Eraser may construct different execution models from the same code and observed execution (i.e., Figure 2.6). Figure 2.7(a) and Figure 2.7(b) show the execution models constructed from Figure 2.6 by FastTrack and Eraser, respectively. In the execution model, operations represent executions of instructions (i.e., reads and writes) which are not synchronizing instructions. An operation p_i denotes the execution of line number i in the example



(a) An execution model of FastTrack (b) An execution model of Eraser

Figure 2.7: Examples of constructed execution models

code. For example, the execution of line 1 (i.e., `x=1`) is represented by an operation p_1=$(th1, write, x)$ which indicates a write operation executed by thread $th1$ on a variable `x`. The rest of the execution can be modeled in a similar way. Executions of synchronization instructions (e.g., lines 1, 4, 11, and 14) are represented by the *synchronized order* relations. If there is a synchronized order between two operations p and q, it means that q always starts only after p finishes.

FastTrack uses the happens-before relation to build a synchronized order relation. In other words, FastTrack builds a synchronized order relation from the order of operations in the same thread (i.e., (p_1, p_3) and (p_{12}, p_{13})) and the order of operations enforced by a lock `m` (i.e.,$(p_3, p_{12}), (p_3, p_{13})$) in the observed execution and their transitive relations (i.e., $(p_1, p_{12}), (p_1, p_{13})$). FastTrack builds an execution model with synchronized order relation $\rightarrow_{S_{FT}}$ as follows:

$$\rightarrow_{S_{FT}} = \{(p_1, p_3), (p_1, p_{12}), (p_1, p_{13}), (p_3, p_{12}), (p_3, p_{13}), (p_{12}, p_{13})\}$$

In contrast, Eraser uses only mutual exclusion constraints induced by lock `m` and the operation order in each thread to generate synchronization order relations. Eraser generates a concurrent execution model with the synchronized order relation $\rightarrow_{S_{ER}}$ as follows:

$$\rightarrow_{S_{ER}} = \{(p_1, p_3), (p_{12}, p_{13}), (p_2, p_{13}), (p_3, p_{13})\}$$

Thus, the data race bug detection results of FastTrack and Eraser can be different because their execution models of the same program and its observed execution can be different as shown above. For the example in Figure 2.6, Eraser reports p_1 and p_{13} as a race bug because p_1 and p_{13} write to x and there is no synchronized order between them in $\rightarrow_{S_{ER}}$. In contrast, FastTrack does not report these two operations as a race bug because there is a synchronized order between these two operations in the execution model (i.e., $\rightarrow_{S_{FT}}$).

Formal Definition of Execution Model

The proposed execution model aims to describe different execution models used by various race bug detection techniques in a uniform manner. For this purpose, the execution model in Definition 1 is *parametric* to race bug detection techniques. For example, the operation in the definition is not restricted to a particular programming language or analysis technique, but general enough to represent all dynamic/static analysis techniques in this survey. In addition, we allow different definitions of *conflict* for different techniques (see Table 2.3).

Definition 1. Execution model

For a race bug detection technique and a target program P with given synchronization requirement specifications (i.e., operation block specification and data association specification), an execution model

is defined as

$$< \Sigma, Thread, Mem, \rightarrow_S, conflict, S_{OB}, S_{DA} >$$

which consists of the following six elements:

- *A set of operations* Σ

  Σ is a set of operations (i.e., $\{p_1, p_2, ...\}$), each of which models a unit of execution of P. An operation p has the following attributes whose concrete definitions depend on a race bug detection technique.

  - $thread(p) \in Thread$ indicates a thread that executes p.
  - $optr(p) \in \{\texttt{read}, \texttt{write}\}$ indicates a type of p, which is either data read or data write.
  - $operand(p) \in Mem$ indicates a memory location or variable manipulated by p (i.e., the operand of the p operation).

- *A set of threads* $Thread$
- *A set of memory locations* Mem
- *A synchronized order relation on operations* $\rightarrow_S$: $\Sigma \times \Sigma$

  $\rightarrow_S$ is a transitive closure of a binary relation on a set of operations $Sync : \Sigma \times \Sigma$ such that $(p, q) \in Sync$ if at least one of the following conditions holds:

  (a) p and q are executed by the same thread and p is executed before q, or

  (b) explicit synchronization causes p to finish before q starts.

  Different race bug detection techniques may generate $\rightarrow_S$ differently (see Section 2.3.2). For example, there are techniques that construct $\rightarrow_S$ using happens-before relations enforced by lock operations in the observed execution (e.g., [21, 33, 35]). Other techniques may generate $\rightarrow_S$ using mutual exclusion relations enforced by lock operations in the observed execution (e.g., [7, 29, 84]). [7]

- *A conflict relation* $conflict : \Sigma \times \Sigma$

  $(p, q) \in conflict$ (i.e., p and q *conflict* with each other) indicates that p and q can interfere each other and cause invalid behaviors. Formally speaking, $(p, q) \in conflict$ holds if the computational effect of p and q depends on the order in which they are executed. A conflict relation should be transitive and symmetric. Again, a concrete definition of $conflict$ may vary depending on the detection technique.

- *A set of operation blocks* $S_{OB} : \Sigma \times \Sigma$

  $(p, p') \in S_{OB}$ indicates that p and p' are executed by the same thread and p executes before p' in an operation block (i.e., $p \rightarrow_S p'$). $(p, p') \in S_{OB}$ indicates that execution from p to p' should not be interfered by another thread.

- *A relation of data association* $S_{DA} : Mem \times Mem$

  S_{DA} indicates a relation on shared variables as described in Section 2.3.1. $(v_1, v_2) \in S_{DA}$ indicates that v_1 and v_2 are parts of a composite data structure and update operations on v_1 and v_2 should be coordinated to satisfy the data structure invariants. Note that S_{DA} is reflexive (i.e., $\forall v \in Mem.(v, v) \in S_{DA}$); S_{DA} may or may not be transitive/symmetric depending on the technique (see Tables 2.4 and 2.5).

□

[7] Note that our execution model represents the *effect* of synchronization with $\rightarrow_S$, not the synchronization itself. Thus, this execution model is general enough to describe various race bug detection techniques with different synchronization mechanisms.

Execution Model Generation

Many race bug detection techniques check the program behavior against race bug definitions as soon as a part of the execution model is constructed (i.e., on-the-fly) for efficiency. In this paper, these two steps are distinguished to describe various techniques in a uniform manner. A total 43 race bug detection techniques are classified (Sections 2.3.3–2.3.6) according to how they generate an execution model.

- Dynamic techniques (34 out of the 43 techniques)
  These techniques instrument the target program to extract concrete values of program executions at runtime, such as thread identifiers, memory addresses, temporal ordering of operations including synchronization, and so on. These techniques can generate an execution model that does not contain infeasible behaviors (i.e., a bug detected by these techniques is always real). However, the generated execution model may miss buggy executions and miss race bugs as a result (i.e., false negatives). Most of the techniques (34 out of 43) in this survey belong to this class [7, 19–21, 33, 35, 39, 53, 54, 57, 59–61, 63, 64, 66, 72, 76, 77, 79, 84, 90, 91, 93, 94, 97, 107, 111–113, 116, 122]. For more detail on how each technique constructs an execution model, see Tables 2.2-2.5.

- Static techniques (7 out of 43 techniques)
  These techniques utilize static analysis methods such as alias analysis [68], data-flow analysis [6], dependency analysis [24], and type systems [31] to obtain the information necessary to construct an execution model. These techniques can generate execution models representing a large scope of possible program executions. However, the accuracy of the resulting execution models is often low due to the imprecision inherent to static analysis methods. Thus, these techniques may raise false positives. In this survey, 7 out of 43 techniques belong to this class [6, 24, 29, 31, 68, 106, 108].

- Hybrid (dynamic + static) techniques (2 out of the 43 techniques)
  HAVE [17] constructs execution models by combining runtime information obtained by dynamic analysis and code information obtained from a static analysis. McPatom [123] also combines dynamic execution information and source code information to generate execution models by considering program semantics.

Most race bug detection techniques generate multiple execution models to check as many thread scheduling scenarios as possible. To generate multiple execution models from a single observed execution (i.e., to predict alternative scheduling scenarios [16]), dynamic techniques mutate orders of operations while satisfying the relations/constraints enforced by synchronization code. Static techniques analyze a target program including synchronization code, to build execution models that represent various thread scheduling scenarios and satisfy the synchronization constraints. These synchronization constraints are used to build a synchronized order relation (i.e., $\rightarrow_S$ in Definition 1).

2.3.3 Data race bug detection techniques

In this section, *data race bug* is defined based on the formal execution model. The data race bug definition does not use the operation block specification nor data association specification, and it is the simplest class of race bugs presented in this paper. This section presents the formal definition of data race bug, and then presents a survey of race bug detection techniques for data race bugs.

Bug Definition

A data race bug is defined as two unsynchronized operations of two different threads that access the same shared variable, where at least one is a write operation. The data race bug definition in this paper coincides with the definition of "data race" in Netzer *et al.* [70]. Many race bug detection techniques [7, 20, 21, 29, 31, 33, 54, 63, 64, 68, 72, 84, 90, 91, 108, 113, 122] target to detect the data race bugs and implement various methods to detect them efficiently (Section 2.3.3). The definition of the data race bug using the formal execution model is as follows:

Definition 2. *Data Race Bug*

A program has a data race bug if an execution model of the program $\langle \Sigma, Thread, Mem, \rightarrow_S, conflict, S_{OB}, S_{DA} \rangle$ *satisfies all of the following conditions:*

There exist two operations $p, q \in \Sigma$ *such that*

DR1: $thread(p) \neq thread(q)$

DR2: $operand(p) = operand(q)$

DR3: $conflict(p, q)$

DR4: $p \not\rightarrow_S q \wedge q \not\rightarrow_S p$

□

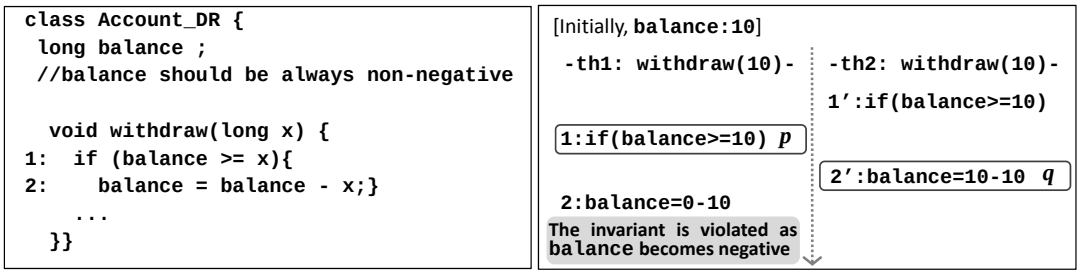
The data race bug definition indicates lack of synchronization on accesses to a shared variable. The DR1 condition checks if p and q are executed by different threads. The DR2 and DR3 conditions together check if p and q may interfere with each other. Note that various techniques have different definitions of $conflict()$ (see the fourth column of Table 2.2). Finally, the DR4 condition checks if p and q can run concurrently without synchronization. In other words, for two operations p and q constituting a data race bug, no synchronization is enforced between p and q (i.e., $p \rightarrow_S q$ or $q \rightarrow_S p$).

Two operations that match the data race bug definition can be harmful because unsynchronized accesses to a shared variable may induce unintended behaviors. Especially, with weak memory models and multicore CPUs, executions depend on runtime status of CPU and shared variable accesses, and often produce counterintuitive results [2, 13]. Enforcing synchronization on all accesses to a shared variable makes the accesses predictable because the access results are consistently determined by an access order. In addition, data race bugs may be harmful even with the sequentially-consistent memory model because a data race bug is caused by unsynchronized concurrent operations on the shared variable, which may not have been intended by developers [8].

Data Race Bug Example

Figure 2.8 shows an example of a data race bug in a buggy program code and an erroneous execution scenario. The example is a bank account program that includes method `withdraw(long x)`. Although `withdraw(long x)` should not withdraw `x` if `x` is larger than `balance` (line 1) to keep `balance` non-negative, the program allows negative `balance` due to the data race bug explained below.

Suppose that, as shown in Figure 2.8(b), there are two threads (`th1` and `th2`) running the bank account program and `balance` is 10 initially. Also, suppose that a data race bug detection technique similar to Eraser [84] is used to generate an execution model from the execution in Figure 2.8(b) with the following elements:



(a) Buggy program code (b) An erroneous execution

Figure 2.8: Data race bug example

- $\Sigma = \{p_1, p_2, p_{1'}, p_{2'}, ...\}$
- $Thread = \{\texttt{th1}, \texttt{th2}\}$
- $operand(p) = operand(q) \in Mem$
- $\rightarrow_S = \{(p_1, p_2), (p_{1'}, p_{2'}), ...\}$
- $(p, q) \in conflict$ if either $optr(p)$ or $optr(q)$ is `write`
- $S_{OB} = \emptyset$
- $S_{DA} = \emptyset$

Operation p (i.e., p_1) of `th1` and operation q (i.e., $p_{2'}$) of `th2` (satisfying DR1) access the same shared variable `balance` (satisfying DR2). In addition, the technique considers that p and q *conflict* with each other (see the fourth column of the fifth row of Table 2.2) because $optr(p)$ is `read` and $optr(q)$ is `write` (satisfying DR3). Furthermore, p and q can run concurrently without synchronization (i.e., $p_1 \not\rightarrow_S p_{2'}$ and $p_{2'} \not\rightarrow_S p_1$ (satisfying DR4)). Since the given execution satisfies all conditions of the data race bug definition, the technique reports a data race bug. This data race bug actually causes an error by assigning -10 to `balance` (line 2 in `th1`), which violates the non-negative invariant on `balance` .

Data Race Bug Detection Techniques

Table 2.2 summarizes 17 techniques for detecting the data race bugs. Each column describes how these techniques check each condition in Definition 2. The four columns (labelled DR1 through DR4 in the top row) are as follows:

- The second column describes mechanisms/methods to obtain thread information of an operation p to check the DR1 condition. The techniques with "runtime thread identifier" [7,20,21,33,54,63,64,72,84, 90,91,113,122] utilize thread identifiers observed in monitored runtime traces to check if two operations are executed by different threads. The techniques with "static approximation" [29, 31, 68, 108] obtain thread identifiers statically, which may produce imprecise results due to the lack of thread sensitivity in static analyses. To obtain a thread identifier, some techniques with "static approximation" try to identify code segments that cannot be executed concurrently, such as interrupt handlers.
- The third column describes the methods to check whether or not p and q may access the same memory location (DR2). The techniques with "code analysis" [29, 31, 68, 108] perform alias analyses on the

Table 2.2: Data race bug detection techniques

Technique	$thread()$ (DR1)	$operand()$ (DR2)	$conflict()$ (DR3)	$\rightarrow_S$ (DR4)
Acculock [113]	runtime thread identifier	runtime memory address	read-write, write-read, write-write	lock
Choi *et al.* [20]	runtime thread identifier	runtime memory address	read-write, write-read, write-write	lock, thread create/join
Chord [68]	static approximation	code analysis	read-read, read-write, write-read, write-write	lock
Eraser [84]	runtime thread identifier	runtime memory address	read-write, write-read, write-write	lock
FastTrack [33]	runtime thread identifier	runtime memory address	read-write, write-read write-write	lock, thread create/join
GUARD [64]	runtime thread identifier	runtime memory address	read-write, write-read, write-write	lock, barrier
LiteRace [63]	runtime thread identifier	runtime memory address	read-write, write-read, write-write	lock, atomic oper. message send/receive
O'Callahan *et al.* [72]	runtime thread identifier	runtime memory address	read-write, write-read, write-write	lock, message send/receive
Racer [7]	runtime thread identifier	runtime memory address	read-write, write-read, write-write	lock, thread create/join
RacerX [29]	static approximation	code analysis	read-write, write-read, write-write	lock
RaceTrack [122]	runtime thread identifier	runtime memory address	read-write, write-read, write-write	lock, thread create/join
RACEZ [91]	runtime thread identifier	runtime memory address	read-write, write-read, write-write	lock
RccJava [31]	static approximation	code analysis	read-read, read-write, write-read, write-write	lock
RELAY [108]	static approximation	code analysis	read-write, write-read, write-write	lock
SOS [54]	runtime thread identifier	runtime memory address	read-write, write-read, write-write	lock, thread create/join
ThreadSanitizer [90]	runtime thread identifier	runtime memory address	read-write, write-read, write-write	lock, message send/receive
TRaDe [21]	runtime thread identifier	runtime memory address	read-write, write-read, write-write	lock

target program code to check if two operations may access the same memory location in any execution. For this purpose, Chord [68] exploits data-flow analyses and RELAY [108] utilizes symbolic executions. RccJava [31] exploits a type system to group operations that may access to the same object.

The techniques with "runtime memory address" [7, 20, 21, 33, 54, 63, 64, 72, 84, 90, 91, 113, 122] use memory locations observed in runtime traces as operands to check if $operand(p)$ is the same as $operand(q)$. Many techniques try to reduce the runtime overhead involved in monitoring all operations. O'Callahan *et al.* [72] leverage static analysis to select statements that may produce data race bugs, and then instrument only those statements to be monitored at runtime. SOS [54] statically finds read-only variables (called *stationary variables*) and then does not monitor accesses to them to save the runtime monitoring cost. Another approach to reduce the runtime monitoring overhead is selecting a subset of memory locations that are likely to be involved in data race bugs to monitor their accesses. LiteRace [63] and ThreadSanitizer [90] utilize the *cold region* hypothesis, which states that frequently tested code segments are less likely to have data race bugs, and exclude frequently executed statements from monitoring. As an alternative approach, RACEZ [91] tries to reduce the monitoring cost by monitoring sampled target memory accesses at the cost of missing data race bugs.

- The forth column describes the methods to check if two operations *conflict* with each other (DR3).

All data race bug detection techniques in Table 2.2 consider a combination of a read operation and a write operation (read-write, or write-read) or a combination of two write operations (write-write) as conflict. RccJava [31] additionally considers the case of two read operations as conflict because the type system of RccJava requires that every operation on a shared variable should be guarded by a lock regardless of the access type.

- The fifth column explains which synchronization mechanisms each technique utilizes to construct the synchronized order relation $\rightarrow_S$ of an execution model to check DR4. All techniques in Table 2.2 utilize observed lock operations (denoted by "lock"). The techniques with "message send/receive" [63,72,122] use message send-and-receive style synchronization as well. The techniques with "thread create/join" [7, 20, 33, 54, 122] additionally use thread operations. LiteRace [63] utilizes atomic operations, and GUARD [64] utilizes barriers in addition to lock operations.

2.3.4 Block race bug detection techniques

Block race bug detection techniques aim to detect race bugs that are caused by unintended concurrent accesses to a shared variable by an operation block and an operation. More specifically, the block race bug detection techniques detect three operations as a block race bug if two operations access the same variable in the same operation block, and the other operation in another thread can access the same variable concurrently to interfere the operation block execution.

Block Race Bug Definition

The block race bug definition is defined in terms of the following three operations: two operations p and p' in an operation block and one operation q that can run concurrently with the two operations. The block race bug definition describes that p and p' in the operation block should not be interfered by the other concurrent operation q for accessing the shared variable m. In the literature, block race bugs are sometimes called as "atomicity violation" [53, 61, 76], or "AI (access interleaving)-invariant violation" [59]. Note that the term "atomicity violation" also refers to a different class of race bugs (i.e., multi-data block race bugs in Section 2.3.6).

The formal definition of the *block race bug* is as follows.

Definition 3. *Block Race Bug*

A program has a block race bug if an execution model of the program $\langle \Sigma, Thread, Mem, \rightarrow_S, conflict, S_{OB}, S_{DA} \rangle$ *satisfies all of the following conditions:*

There exist three operations $p, p', q \in \Sigma$ *such that*

BR1: $thread(p) = thread(p')$

BR2: $(p, p') \in S_{OB}$

BR3: $operand(p) = operand(p')$

BR4: $thread(p) \neq thread(q)$

BR5: $operand(p) = operand(q)$

BR6: $conflict(p, q) \wedge conflict(q, p')$

BR7: $q \not\rightarrow_S p \quad \wedge \quad p' \not\rightarrow_S q$

□

The BR1–BR3 conditions check if two operations p and p' of the same thread (BR1) access the same variable (BR3) in an operation block (BR2). Note that for two operations p and p' that satisfy BR2, p is executed before p' in the operation block (i.e., $p \rightarrow_S p'$).

The BR4–BR7 conditions check if operation q can run concurrently with p and p', and access the same variable. The BR4 and BR7 conditions check if q is executed by a different thread from the thread of p and p' (BR4) and q can be executed while a thread executes p and p' in an operation block (i.e., if q can execute between p and p') (BR7). Note that q does not have to belong to any operation block. The BR5–BR6 conditions check if p and q access the same variable, and thus, potentially interfere each other (similarly for q and p').

Note that the BR4–BR7 conditions are similar to the DR1–DR4 conditions for the data race bug definition, while BR4–BR7 have additional constraints for the relation between q and p'. In Section 2.3.7, a detailed comparison of the block race bugs with the data race bugs is presented.

Block Race Bug Example

Figure 2.9 shows an example of a program with a block race bug and an erroneous execution due to the block race bug. Note that the program should keep `balance` non-negative. Lines 4–8 of method `void withdraw(long x)` are specified as an atomic region, thus the operations that execute lines 4–8 are defined as an operation block. Suppose that there are two threads (`th1` and `th2`) running the bank account program in Figure 2.9 and `balance` is initially 10.

Suppose that a block race bug detection technique like PENELOPE [94] (see the 11th row of Table 2.3) is applied to Figure 2.9. Operations p and p' are executed by the `th1` thread (satisfying BR1) in the same operation block (satisfying BR2) and access the same variable `balance` (satisfying BR3) (see Figure 2.9(b)).

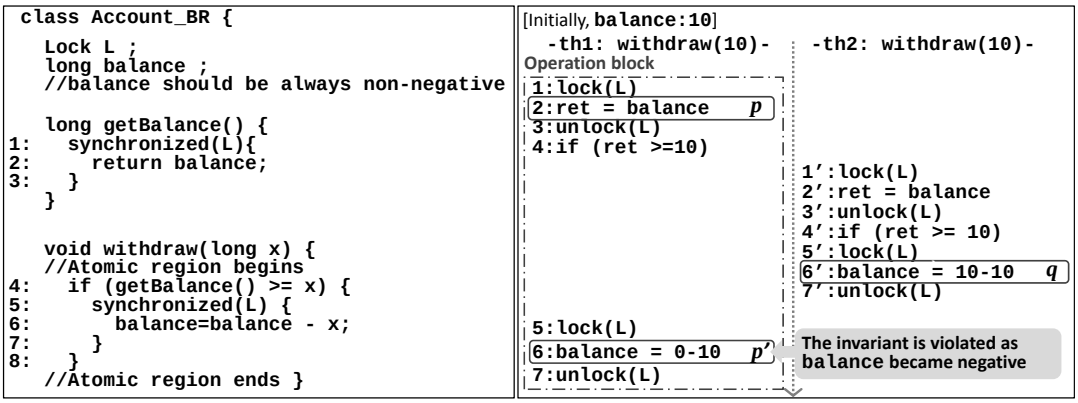
The `th2` thread executes q (satisfying BR4) that accesses the same variable `balance` (satisfying BR5). Operations p and q conflict with each other and operations q and p' have a conflict as well (satisfying BR6) because $optr(p)$ is `read` while $optr(q)$ is `write` , and $optr(q)$ is `write` while $optr(p')$ is `write` (see the fourth column of the 11th row of Table 2.3). Finally, q occurs between p and p' (satisfying BR7). Therefore, the execution model for the given execution satisfies all conditions of the block race bug definition, and thus, the technique detects a block race bug. This execution actually results in an error by assigning -10 to `balance` (line 6 in `th1`), which violates the intention of the program. The error occurs because the condition check on `balance` (line 4) and the following update of `balance` (line 6) are not guarded by the same synchronized block, which allows a concurrent update of `balance` (line 6) by the other thread.

Block Race Bug Detection Techniques

Table 2.3 summarizes the ten block race bug detection techniques [19, 53, 59, 61, 76, 77, 93, 94, 123]. [8] The four columns of the table are as follows:

- The second column describes the methods to check whether two operations are executed by the same thread or different threads (BR1 and BR4). All block race bug detection techniques in this survey utilize thread identifier information observed in actual traces to check the BR1 and BR4 conditions.
- The third column shows the methods to check if two operations access the same variable (BR3 and BR5). In this survey, all techniques compare concrete memory locations observed in actual traces.

[8] Our survey does not include Atomizer [32] because Atomizer checks locking patterns rather than data access patterns.



(a) Buggy program code (b) An erroneous execution

Figure 2.9: Block race bug example

- The fourth column shows the criteria to determine if two operations executed by different threads conflict with each other (BR6). The three techniques [77,94,123] consider that two operations conflict with each other when at least one operation is a write operation, which is denoted as "read-write, write-read, write-write" in the table. The other seven techniques [19, 53, 59, 61, 76, 93] consider p and q (or p' and q) as conflicting with each other if they are "read-write, write-read, write-write′" on a variable. These techniques consider p and q as not conflicting if they write to a variable m and the operation block of p has another operation that overwrites m before any read operation (denoted by write-write′). The techniques with "write-write" conflict follow the notion of *conflict-serializability*. The techniques with "write-write′" conflict follow the notion of *view-serializablility*. [9]
- The last column describes the synchronization mechanisms to construct the synchronized order relation $\rightarrow_S$ to check BR7. For this purpose, six techniques [53,61,76,94,112,123] track lock operations (denoted by "lock"). The block-based algorithm [112] additionally uses message send/receive operations, and CTrigger [76] and McPatom [123] consider other synchronization mechanisms including wait-and-notify, thread creation/join, and barrier to generate more accurate execution models. The other four techniques [19, 59, 77, 93] approximately check BR7 based on the observed temporal orders of operations (denoted by "timestamp") without considering explicit synchronization operations (i.e., check if starting time instances of p, q and p' are in sequence). These four techniques may miss a block race bug when operation q happens to be executed before p or after p'.

2.3.5 Multi-data race bug detection techniques

The multi-data race bug detection techniques target a race bug that inconsistently updates two associated variables specified in the data association specification S_{DA}. This type of race bugs is characterized by unsynchronized concurrent accesses to the associated variables.

Multi-Data Race Bug Definition

Definition 4. *Multi-data Race Bug*

[9] An in-depth comparison of the algorithmic complexity and bug detection precision for these two notions is found in Qadeer *et al.* [80].

Table 2.3: Block race bug detection techniques

Technique	$thread()$ (BR1, BR4)	$operand()$ (BR3, BR5)	$conflict()$ (BR6)	$\rightarrow_S$ (BR7)
Atom-Aid [61]	runtime thread identifier	runtime memory address	read-write, write-read, write-write′	lock
AtomRace [53]	runtime thread identifier	runtime memory address	read-write, write-read, write-write′	lock
AVIO [59]	runtime thread identifier	runtime memory address	read-write, write-read, write-write′	timestamp
Block-based algorithm [112]	runtime thread identifier	runtime memory address	read-write, write-read, write-write′	lock, message send/receive
CTrigger [76]	runtime thread identifier	runtime memory address	read-write, write-read, write-write′	barrier, lock, thread create/join
DefUse [93]	runtime thread identifier	runtime memory address	read-write, write-read, write-write	timestamp
FALCON [77]	runtime thread identifier	runtime memory address	read-write, write-read, write-write	timestamp
Kivati [19]	runtime thread identifier	runtime memory address	read-write, write-read, write-write′	timestamp
McPatom [123]	runtime thread identifier	runtime memory address	read-write, write-read, write-write	lock, wait/notify
PENELOPE [94]	runtime thread identifier	runtime memory address	read-write, write-read, write-write	lock

A target program has a multi-data race bug if an execution model of the program $\langle \Sigma, Thread, Mem, \rightarrow_S, conflict, S_{OB}, S_{DA} \rangle$ *satisfies all of the following conditions:*

There exist two operations $p, q \in \Sigma$ *such that*

MR1: $thread(p) \neq thread(q)$

MR2: $(operand(p), operand(q)) \in S_{DA}$

MR3: $conflict(p, q)$

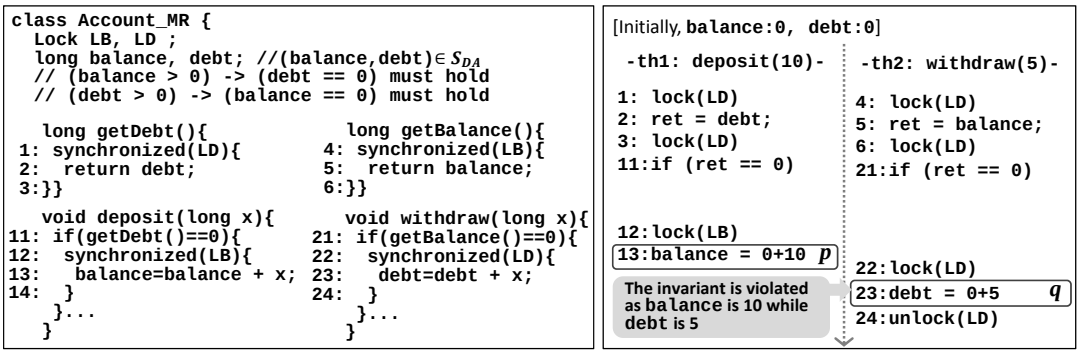
MR4: $p \not\rightarrow_S q \ \wedge \ q \not\rightarrow_S p$

□

The MR1, MR3 MR4 conditions are the same as the DR1, DR3, and DR4 conditions of the data race bug, respectively. The MR2 condition utilizes data association S_{DA} to check if p and q access associated variables. Section 2.3.7 presents a detailed comparison of the multi-data race bugs and the data race bugs.

Multi-data Race Bug Example

Figure 2.10 describes an example of a program containing a multi-data race bug and an erroneous execution caused by the bug. In the example, `balance` and `debt` represent the amount of balance and the amount of debt in a bank account, respectively. The program has an invariant that `debt` is zero when `balance` is positive (i.e., `(balance>0)`→`(debt==0)`) and `balance` is zero when `debt` is positive (i.e., `(debt>0)`→ `(balance==0)`). Therefore, `balance` and `debt` are associated and should be updated consistently to satisfy the invariant. The `deposit(long x)` method increases `balance` by `x` when `debt` is zero (lines 11–14). The `withdraw(long x)` method increases `debt` by `x` when `balance` is zero (lines 21–24).



(a) Buggy program code (b) An erroneous execution

Figure 2.10: Multi-data race bug example

Suppose that a multi-data race bug detection technique similar to the object race detection technique [107] is used (see the fourth row of Table 2.4). Suppose that there are two threads, `th1` and `th2`, running the bank account program where `balance` is 0 and `debt` is 10 initially. Also, suppose that `balance` and `debt` are associated to satisfy the invariant. Operation p on `th1` and operation q on `th2` (satisfying MR1) access `balance` and `debt` respectively (satisfying MR2 as $(\texttt{balance}, \texttt{debt}) \in S_{DA}$). In addition, p and q conflict with each other, since $optr(p)$ is write and $optr(q)$ is also write (satisfying MR3). Furthermore, p and q run concurrently without synchronization (satisfying MR4).

Therefore, the given execution scenario satisfies all conditions of the multi-data race bug definition, and thus, the multi-data race is detected. This execution actually causes an error, since `balance` becomes 10 and `debt` becomes 5, which violates the invariant. In other words, the interleaved execution updates associated variables inconsistently because p and q are protected by different locks.

Multi-data Race Bug Detection Techniques

Table 2.4 gives an overview of the multi-data race bug detection techniques. The columns represent the following:

- The second column shows that all three multi-data race bug detection techniques in this survey [57,79, 107] use runtime thread identifiers observed in actual traces to check if two operations are executed by the same thread or not (for checking MR1).
- The third to sixth columns (MR2) represent the type and characteristics of the data association relation for each technique. Data association for a target execution model is specified as a relation over two variables. The third to fifth columns show whether or not the data association relation is reflexive (i.e., $\forall a \in Mem.(a,a) \in S_{DA}$), symmetric (i.e., $\forall a,b \in Mem.(a,b) \in S_{DA} \rightarrow (b,a) \in S_{DA}$), or transitive (i.e., $\forall a,b,c \in Mem.(a,b) \in S_{DA} \wedge (b,c) \in S_{DA} \rightarrow (a,c) \in S_{DA}$), respectively. The sixth column ("Type") shows the type of the data association relation used by the technique. MultiRace [79] and the object data race detection technique [107] construct data association relation as a set of two memory locations. MUVI-Eraser [57] defines a data association as an ordered pair of variables (i.e., not symmetric) with conditions on the access types to the variables (i.e., read or write).

  In addition, all three techniques in this category utilize memory locations observed in actual traces to obtain operand information and check MR2.
- The seventh column shows that all three techniques in this class consider two operations as conflicting if they access the associated variables and at least one operation is a write operation (MR3).

Table 2.4: Multi-data race bug detection techniques

Technique	$thread()$ (MR1)	S_{DA} (MR2)				$conflict()$ (MR3)	$\rightarrow_S$ (MR4)
		Ref	Sym	Tran	Type		
MultiRace [79]	runtime thread identifier	○	○	○	pairs of mem. locations	read-write, write-read, write-write	lock, thread create/join
MUVI-Eraser [57]	runtime thread identifier	○	×	×	ordered pairs of mem. locations w/ access types	read-write, write-read, write-write	lock
Object data race detection [107]	runtime thread identifier	○	○	○	pairs of mem. locations	read-write, write-read, write-write	lock

- The last column represents that all three techniques track lock operations to build a synchronized order relation $\rightarrow_S$ to check MR4. MultiRace [79] additionally considers thread creation and join.

The methods that these techniques employ to obtain data association are as follows. MultiRace [79] aggregates continuous shared memory locations allocated for a composite data structure into one cluster and considers any two memory locations in the cluster as associated. Consequently, the data association relation becomes an equivalence relation (i.e, transitive, symmetric, and reflexive). Similarly, object data race detection technique [107] considers members/fields of an object as associated data. These two techniques obtain the data association specification from data structures declared in a target program. MUVI [57] utilizes data association specification inferred by a heuristic that considers two variables as associated if they are frequently accessed in close distance.

2.3.6 Multi-data block race bug detection techniques

This section presents the definition of *multi-data block race bug*, and 13 techniques that target the multi-data block race bug. A multi-data block race bug consists of two operations within an operation block and another operation that runs concurrently with the two operations that access associated variables in unsynchronized manner. The race bug detection techniques in this category detect multi-data block race bugs that violate combined specifications of operation blocks and data associations.

Multi-data Block Race Bug Definition

Definition 5. *Multi-data Block Race Bug*

A program has a multi-data block race bug if an execution model of the program $\langle \Sigma, Thread, Mem, \rightarrow_S, conflict, S_{OB}, S_{DA} \rangle$ *satisfies all of the following conditions:*

There exist three operations $p, p', q \in \Sigma$ *such that*

MBR1: $thread(p) = thread(p')$

MBR2: $(p, p') \in S_{OB}$

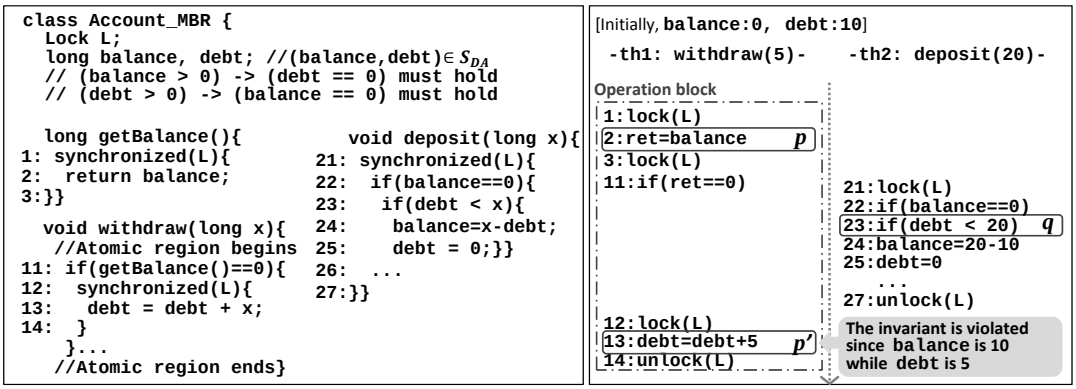
MBR3: $(operand(p), operand(p')) \in S_{DA}$

MBR4: $thread(p) \neq thread(q)$

MBR5: $(operand(p), operand(q)) \in S_{DA} \wedge (operand(q), operand(p')) \in S_{DA}$

MBR6: $conflict(p, q) \wedge conflict(q, p')$

MBR7: $q \not\rightarrow_S p \wedge p' \not\rightarrow_S q$



(a) Buggy program code (b) An erroneous execution

Figure 2.11: Multi-data block race bug example

□

The MBR1–MBR3 conditions specify that operations p and p' should be executed by the same thread (MBR1) within the same operation block (MBR2), and the $operand(p)$ and $operand(p')$ are data associated (MBR3). The MBR4–MBR7 conditions check if there exists an operation that can be executed by another thread (MBR4), between p and p' (MBR7), and the operation can interfere with the data association of p and p' (MBR5 and MBR7).

There exist techniques that have additional condition for MBR5. The five techniques [17,35,66,111, 116] additionally check if the operands of p and q (or the operands of q and p') should be the same. Other two techniques [5, 106] detect three operations as a multi-data race bug only if at least two associated variables are involved with the three operations.

The multi-data block race bug definition is an extension of the block race bug definition (Definition 3) to check additional conditions on data association relations, in a way similar to the multi-data race bug definition (Definition 4). The MBR1, MBR2, MBR4, MBR6, and MBR7 conditions are identical to the BR1, BR2, BR4, BR6, and BR7 conditions in the block race bug definition. The MBR3 and MBR6 conditions are adapted from MR2 to check if variables accessed by two operations are associated. Section 2.3.7 presents a detailed comparison of block race bug and multi-data block race definitions. Similarly, the comparison of the multi-data race bug definition with the multi-data block race bugs is presented in Section 2.3.7.

Multi-data Block Race Bug Example

Figure 2.11 shows an example of a multi-data block race bug. The concurrent execution causes an error because operations p and p' that access two associated variables (`balance` and `debt`) should be synchronized as a whole but separately. Thus, because of the intervening q of `th2`, the values of `balance` and `debt` are updated inconsistently and violate the invariant.

Multi-data Block Race Detection Techniques

Table 2.5 is an overview of 13 multi-data block race detection techniques.

- The second column shows that the ten techniques with "runtime thread identifier" use the information of thread identifiers observed in actual traces to check the MBR1 condition [5, 17, 35, 39, 57, 60,

66, 97, 111, 116]. The other three techniques [6, 24, 106] consider that two operations of two public methods/functions can be executed by different threads.

- The third to sixth columns show the properties of data association relation S_{DA} in each technique (for checking MBR3 and MBR5). The data association relations used by the three techniques [39, 60, 97] are reflexive, transitive, and symmetric. Thus, the data association relation of an execution model can be represented as a set of partitions of memory locations. For example, an *atomic-set* [39, 104] defines a set of member fields of an object as associated. The sixth column shows that the three techniques [24, 57, 106] generate the data association relation by specifying two variables with access types (read or write).
- The seventh column represents the method to obtain operand information for each technique to check the MRB3 and MBR5 conditions. The eight techniques with "runtime mem." [35, 39, 57, 60, 66, 97, 111, 116] use runtime memory addresses observed in actual traces. The four techniques with "code analysis" use static analysis to estimate operand information. HAVE [17] utilizes a hybrid method that use both runtime trace and code analysis result.
- The eighth column shows that the four techniques [5, 6, 24, 106] consider that two operations accessing associated variables conflict with each other (MBR6) regardless of their types. Other eight techniques [17, 35, 39, 60, 66, 97, 111, 116] consider two operations as conflicting, when at least one operation is a write operation. MUVI-AVIO [57], an extension of AVIO [59], does not consider two write operations p and q as conflicting, when another write operation in the same operation block as p overwrites the variable before any read (denoted by write-write′).
- The last column shows that the eight techniques with "lock" [5,6,17,24,35,106,111,116] obtain synchronized order relations by tracking lock synchronization (MBR7). In addition to tracking lock ordering, COPPER [116] considers thread creation/join and wait/notify. The commit-node algorithm [111] and HAVE consider the execution order induced by message send/receive, to build the synchronized order $\rightarrow_S$. The five techniques with "timestamp" [39, 57, 60, 66, 97] use the observed temporal ordering of operations to generate the synchronization order relation without considering explicit synchronization instructions (i.e., $p \rightarrow_S q$ if $timestamp(p) < timestamp(q)$). Since these methods over-approximate the synchronization relation, the five techniques may miss existing multi-data block race bugs.

The multi-data block race bug detection techniques use different methods to obtain data association specification. First, the three techniques [39, 60, 97] ask users to concretely specify the data association specification. The other techniques use program analysis or heuristics to infer data association relations from the target program code or observed runtime traces. Dias *et al.* [24] analyze the target program code to identify control dependency and data dependency relation in shared variable accesses, and then consider two variables with data dependency or control dependency as associated. MUVI-AVIO [57] infers likely data association relations by analyzing statistics on the variable accesses in the target program code. The other techniques [5, 6, 17, 24, 35, 66, 106, 111, 116] consider two variables accessed in an operation block as data associated. The block-local atomicity technique [6] defines two variables `x` and `y` as associated when the two variables are accessed in the same method and there exists a local variable that is assigned by the value of `x` and then used to define `y`.

2.3.7 Relations between race bug classes

Based on the formal definitions of the four classes of race bugs, it is possible to clarify the relation among the four classes of race bugs using the following notations:

Table 2.5: Multi-data block race detection techniques

Technique	$thread()$ (MBR1, MBR4)	S_{DA} (MBR3, MBR5)				$operand()$ (MBR3, MBR5)	$conflict()$ (MBR6)	$\rightarrow_S$ (MBR7)
		Ref	Sym	Tran	Type			
Atomic-set serial. violation detector [39]	runtime thread identifier	○	○	○	pairs of mem. locations	runtime mem.	read-write, write-read, write-write	timestamp
AtomTracker [66]	runtime thread identifier	○	○	×	pairs of mem. locations	runtime mem	read-write, write-read, write-write	timestamp
Block-local atomicity [6]	static approximation	○	×	×	ordered pairs of mem. locations	code analysis	read-read, read-write, write-read, write-write	lock
ColorSafe [60]	runtime thread identifier	○	○	○	pairs of mem. locations	runtime mem.	read-write, write-read, write-write	timestamp
Commit-node [111]	runtime thread identifier	○	○	×	pairs of mem. locations	runtime mem.	read-write, write-read write-write	lock, message send/receive
COPPER [116]	runtime thread identifier	○	○	×	pairs of mem. locations	runtime mem.	read-write, write-read, write-write	lock, wait/notify, thread create/join
Dias *et al.* [24]	static approximation	○	×	×	ordered pairs of mem. locations w/ access types	code analysis	read-read, read-write, write-read, write-write	lock
HAVE [17]	runtime thread identifier	○	○	×	pairs of mem. locations	runtime mem., code analysis	read-write, write-read write-write	lock, message send/receive
High-level data race detection [5]	runtime thread identifier	○	○	×	pairs of mem. locations	code analysis	read-read, read-write, write-read, write-write	lock
Marathon [97]	runtime thread identifier	○	○	○	pairs of mem. locations	runtime mem.	read-write, write-read, write-write	timestamp
Method-consistency [106]	static approximation	○	○	×	pairs of mem. locations w/ access type	code analysis	read-read, read-write, write-read, write-write	lock
MUVI-AVIO [57]	runtime thread identifier	○	×	×	ordered pairs of mem. locations w/ access type	runtime mem.	read-write, write-read, write-write′	timestamp
Velodrome [35]	runtime thread identifier	○	○	×	pairs of mem. locations	runtime mem.	read-write, write-read, write-write	lock

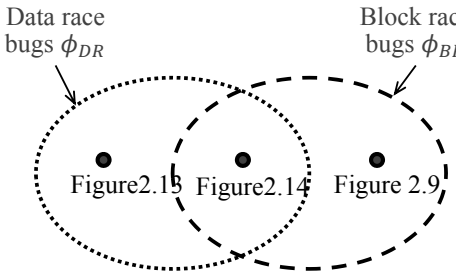
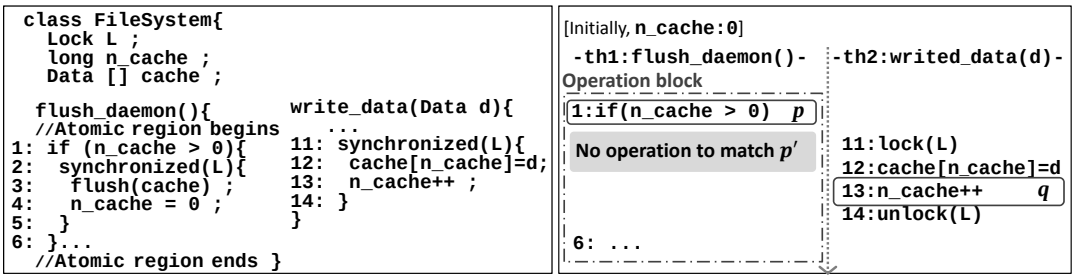


Figure 2.12: Relation between data race bugs and block race bugs

- F_α denotes a conjunctive formula of all conditions of α race bug definition, where $\alpha \in$ {DR, BR, MR, MBR} representing classes of data race bugs, block race bugs, multi-data race bugs, and multi-data block race bugs, respectively.
- f_c denotes a condition c of a race bug definition, where $c \in$ {DR1, DR2, DR3, DR4}$\cup${BR1, BR2, BR3, BR4, BR5, BR6, BR7}$\cup${MR1, MR2, MR3, MR4}$\cup${MBR1, MBR2, MBR3, MBR4, MBR5, MBR6, MBR7}
- ϕ_α denotes the set of execution models satisfying F_α. For every execution model σ, $\sigma \models F_\alpha$ if and only if $\sigma \in \phi_\alpha$. ϕ_α^C denotes the complement of the set ϕ_α. ϕ_α^C defines the set of execution models that satisfies $\neg F_\alpha$.



(a) Buggy program code (b) An erroneous execution

Figure 2.13: Example of a data race bug not detected by block race bug detectors

Data race bugs vs. block race bugs

The set of execution models that satisfy the data race bug condition (i.e., ϕ_{DR}) overlaps with the set of execution models that satisfy the block race bug conditions (i.e., ϕ_{BR}). Figure 2.12 describes the relation of ϕ_{DR} and ϕ_{BR}, and the examples correspond to each region of the diagram. We can prove the relation by showing that $\phi_{DR} \not\subseteq \phi_{BR}$, $\phi_{DR} \cap \phi_{BR} \neq \emptyset$, and $\phi_{BR} \not\subseteq \phi_{DR}$.

Theorem 1. $\phi_{DR} \not\subseteq \phi_{BR}$

Proof. $\phi_{DR} \not\subseteq \phi_{BR}$ *holds if and only if* $\phi_{DR} \cap \phi_{BR}^{C} \neq \emptyset$. $\phi_{DR} \cap \phi_{BR}^{C} \neq \emptyset$ *holds if and only if* $F_{DR} \wedge \neg F_{BR}$ *is satisfiable (i.e., there exists an execution model* σ *such that* $\exists p, q, p' \in \Sigma. F_{DR} \wedge \neg F_{BR}$*). We can rewrite the formula as follows:* $F_{DR} \wedge \neg(f_{BR4} \wedge f_{BR5} \wedge f_{BR1} \wedge f_{BR2} \wedge f_{BR3} \wedge f_{BR6} \wedge f_{BR7})$*. This formula is satisfiable if and only if there exists an execution model that satisfies* $F_{DR} \wedge \neg(f_{BR1} \wedge f_{BR2} \wedge f_{BR3} \wedge f_{BR6} \wedge f_{BR7})$ *because* $F_{DR} \wedge \neg(f_{BR4} \wedge f_{BR5})$ *is unsatisfiable as* $f_{DR1} = f_{BR4}$ *and* $f_{DR2} = f_{BR2}$*. Thus,* $\phi_{DR} \not\subseteq \phi_{BR}$ *holds if there exists an execution model that has two operations* p *and* q *which satisfy the data race bug definition* F_{DR}*, while the execution model has no operation for* p' *in the operation block that contains* p*.*

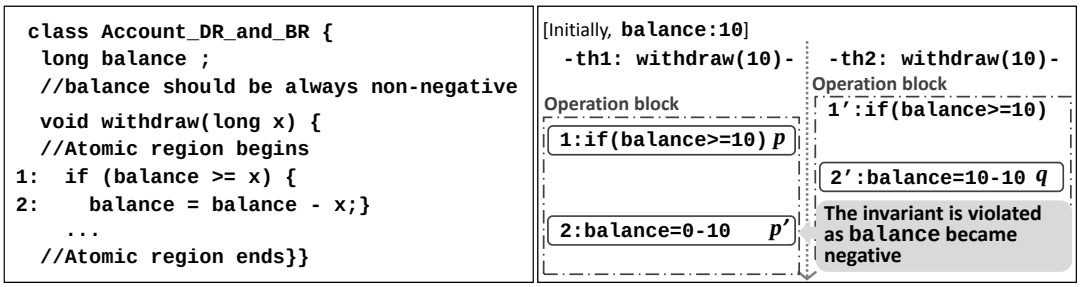
Figure 2.13 illustrates a program with an execution scenario which has a data race bug but no block race bug. A data race bug detection technique such as Eraser [84] detects a data race bug for p *and* q *because these read and write the variable* `n_cache` *without synchronization. However, block race bug detection techniques such as AVIO [59] do not detect any block race bug for* p *and* q *because the execution model has no operation to match* p'*.*

$\phi_{DR} \cap \phi_{BR} \neq \emptyset$

Proof. $\phi_{DR} \cap \phi_{BR} \neq \emptyset$ *holds if and only if the following formula is satisfiable:* $F_{DR} \wedge F_{BR}$*. This formula is equivalent to* $F_{DR} \wedge f_{BR1} \wedge f_{BR2} \wedge f_{BR3} \wedge f_{BR6} \wedge f_{BR7}$ *as DR1 = BR4 and DR2 = BR5. Figure 2.14 describes an example that belongs to the intersection of* ϕ_{DR} *and* ϕ_{BR}*, which extends the example of Figure 2.8 by adding an operation block specification. A block race bug detection technique can detect operations* p*,* p' *and* q *as a race bug because* p *and* p' *belong to the same operation block of thread* `th1` *(satisfying BR1 and BR2), all three operations access the same variable* `balance` *(satisfying BR3),* q *and* p' *conflict with each other as both operations are writing (satisfying BR6), and* q *can run concurrently with* p *and* p' *(satisfying BR7).*

$\phi_{BR} \not\subseteq \phi_{DR}$

Proof. $\phi_{BR} \not\subseteq \phi_{DR}$ *holds if and only if* $\phi_{BR} \cap \phi_{DR}^{C} \neq \emptyset$*. The relation holds if and only if the following formula is satisfiable:* $F_{BR} \wedge \neg F_{DR}$*. The formula is equivalent to the formula* $F_{BR} \wedge \neg f_{DR4}$ *because DR1*



(a) Buggy program code (b) An erroneous execution

Figure 2.14: Example of a data race bug detected by block race bug detectors

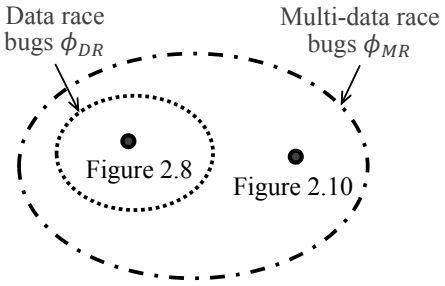


Figure 2.15: Relation between data race bugs and multi-data race bugs

= BR4, DR2 = BR5, and BR6 implies DR3 (i.e., $\forall\sigma.\sigma \models f_{BR6} \Rightarrow \sigma \models f_{DR3}$). Figure 2.9 shows an example which has a block race bug but no data race bug. The data race bug detection techniques do not detect the data race bug in Figure 2.9 since p and q are synchronized (similarly, q and p′). □

The practical implication of Theorem 1 is that it is a good idea to use data race bug detectors and block race bug detectors together as complements since data race bug detectors cannot detect all block race bugs, and vice versa.

Data race bugs vs. multi-data race bugs

Figure 2.15 shows that the class of data race bugs is a proper subset of the class of multi-data race bugs. We can prove this relationship using the race bug definitions.

Theorem 2. $\phi_{DR} \subset \phi_{MR}$

Proof. $\phi_{DR} \subseteq \phi_{MR}$ *holds if and only if* $\phi_{DR} \cap \phi_{MR}^{C} = \emptyset$. $\phi_{DR} \cap \phi_{MR}^{C} = \emptyset$ *holds if and only if* $F_{DR} \wedge \neg F_{MR}$ *is unsatisfiable.* $F_{DR} \wedge \neg F_{MR}$ *is unsatisfiable if* $f_{DR2} \wedge \neg f_{MR2}$ *is unsatisfiable because* $DR1 = MR1$, $DR3 = MR3$, *and* $DR4 = MR4$. $DR2$ *implies* $\exists v.(v,v) \in S_{DA} \wedge v = operand(p) = operand(q)$ *(since* $\forall v.(v,v) \in S_{DA}$*). Note that if* $opernad(p) = operand(q)$, $f_{MR2} = \exists v.(v,v) \in S_{DA} \wedge v = operand(p) = operand(q)$. *Thus,* $f_{DR2} \wedge \neg f_{MR2}$ *is unsatisfiable and* $F_{DR} \wedge \neg F_{MR}$ *is unsatisfiable. Thus,* $\phi_{DR} \cap \phi_{MR}^{C} = \emptyset$ *and* $\phi_{DR} \subseteq \phi_{MR}$ *hold.*

In addition, it is clear that there exists an execution model that satisfies the multi-data race bug definition but not the data race bug definition (for example, Figure 2.10). Thus, the class of data race bugs is a proper subset of the class of multi-data race bugs.

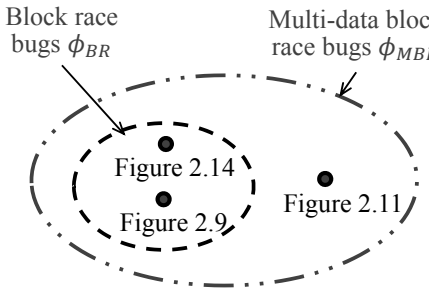


Figure 2.16: Relationship between block race bugs and multi-data block race bugs

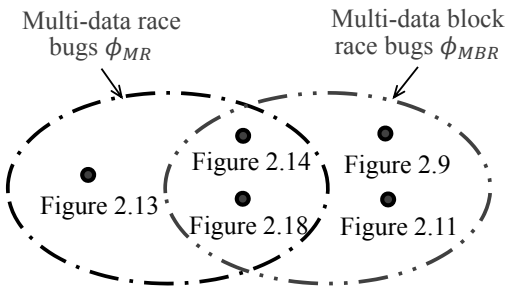


Figure 2.17: Relationship between multi-data race bugs and multi-data block race bugs

□

Block race bugs vs. multi-data block race bugs

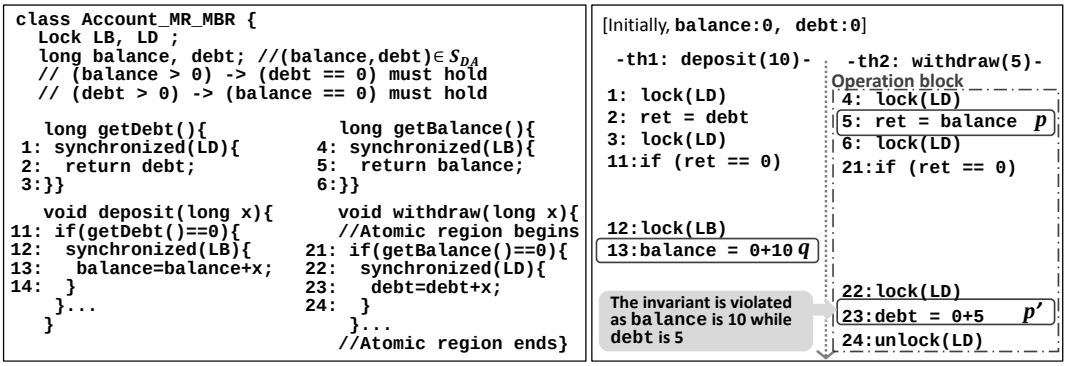
Figure 2.16 shows that the class of the block race bug is a proper subset of the class of the multi-data block race bug. We can prove this relation using the race bug definitions.

Theorem 3. $\phi_{BR} \subset \phi_{MBR}$

Proof. $\phi_{BR} \subseteq \phi_{MBR}$ *holds if and only if* $\phi_{BR} \cap \phi^C_{MBR} = \emptyset$. $\phi_{BR} \cap \phi^C_{MBR} = \emptyset$ *holds if and only if* $F_{BR} \wedge \neg F_{MBR}$ *is unsatisfiable. As BR1 = MBR1, BR2 = MBR2, BR4 = MBR4, BR6 = MBR6, and BR7 = MBR7,* $F_{BR} \wedge \neg F_{MBR}$ *is unsatisfiable if* $(f_{BR3} \wedge f_{BR5} \wedge \neg f_{MBR3}) \vee (f_{BR3} \wedge f_{BR5} \wedge \neg f_{MBR5})$ *is unsatisfiable.* $BR3$ *indicates* $\exists v.(v, v) \in S_{DA} \wedge v = operand(p) = operand(q)$ *(since* $\forall v.(v, v) \in S_{DA}$*). Note that if* $operand(p) = operand(q)$, $f_{MBR3} = \exists v.(v, v) \in S_{DA} \wedge v = operand(p) = operand(q)$. *Thus,* $f_{BR3} \wedge \neg f_{MBR3}$ *is unsatisfiable and* $f_{BR5} \wedge \neg f_{MBR5}$ *is also unsatisfiable for the similar reason. As a result,* $F_{BR} \wedge \neg F_{MBR}$ *is unsatisfiable.*

In addition, it is clear that there exists an execution model that satisfies a multi-data block race bug definition but not the block race bug definition (for example, Figure 2.11). Thus, the class of block race bugs is a proper subset of the class of multi-data block race bugs.

□



(a) Buggy program code (b) An erroneous execution

Figure 2.18: Example of a multi-data race bug detected by multi-data block race bug detectors

Multi-data race bugs vs. multi-data block race bugs

The class of the multi-data race bug ϕ_{MR} overlaps the class of the multi-data block race bug ϕ_{MBR} as shown in Figure 2.17. We can prove the relation of ϕ_{MR} and ϕ_{MBR} by showing that $\phi_{MR} \not\subseteq \phi_{MBR}$, $\phi_{MR} \cap \phi_{MBR} \neq \emptyset$, and $\phi_{MBR} \not\subseteq \phi_{MB}$.

Theorem 4. $\phi_{MR} \not\subseteq \phi_{MBR}$

Proof. $\phi_{MR} \not\subseteq \phi_{MBR}$ *holds if and only if* $F_{MR} \wedge \neg F_{MBR}$ *is satisfiable. The formula is equivalent to* $F_{MR} \wedge \neg(f_{MBR1} \wedge f_{MBR2} \wedge f_{MBR3} \wedge f_{MBR7})$ *because MBR4 = MR1, MBR5 implies MR2 (i.e.,* $\forall\sigma.\sigma \models f_{MBR5} \Rightarrow \sigma \models f_{MR2}$*), and MBR6 implies MR3 (i.e.,* $\forall\sigma.\sigma \models f_{MBR6} \Rightarrow \sigma \models f_{MR3}$*). Figure 2.13 is an example that has a multi-data race bug, but no multi-data block race bug since no operation matches* p' *in the operation block of* p*.*

$\phi_{MR} \cap \phi_{MBR} \neq \emptyset$

Proof. We can prove this statement by using Theorems 1, 2, and 3. $\phi_{DR} \subseteq \phi_{MR}$ *(Theorem 2) and* $\phi_{BR} \subseteq \phi_{MBR}$ *(Theorem 3) imply that* $\phi_{DR} \cap \phi_{BR} \subseteq \phi_{MR} \cap \phi_{MBR}$*. Thus,* $\phi_{MR} \cap \phi_{MBR} \neq \emptyset$ *because* $\phi_{DR} \cap \phi_{BR} \neq \emptyset$ *(Theorem 1). Figure 2.18 describes the case for* $\phi_{MR} \cap \phi_{MBR}$*, which has the same code and execution as Figure 2.10. The multi-data race bug detectors detect two operations* p *and* q *as a multi-data race bug because these operations access* `balance` *and* `debt` *from different threads without synchronization. In addition, the multi-data block race detection techniques can detect operations* p*,* p' *and* q *as a multi-data block race bug because these operations manipulate associated variables* `balance` *and* `debt` *without synchronization.*

$\phi_{MBR} \not\subseteq \phi_{MR}$

Proof. $\phi_{MBR} \not\subseteq \phi_{MR}$ *holds if and only if there exists an execution model that satisfies* $F_{MBR} \wedge \neg F_{MR}$*. This formula is satisfiable if and only if* $F_{MBR} \wedge \neg f_{MR4}$ *because MBR4 = MR1, MBR5 implies MR2 (i.e.,* $\forall\sigma.\sigma \models f_{MBR5} \Rightarrow \sigma \models f_{MR2}$*), and MBR6 implies MR3 (i.e.,* $\forall\sigma.\sigma \models f_{MBR6} \Rightarrow \sigma \models f_{MR3}$*). Figure 2.11 shows a case where the generated execution model has a multi-data block race bug involving* p, q, p'*, but no multi-data race bug. Since operations* p, p', q *are ordered with synchronization, multi-data race bug detectors cannot detect these as a race bug.* □

2.3.8 Other work on race bug survey

There are a few survey papers on race bugs and race bug detection techniques. Netzer *et al.* [70] formalizes two classes of race conditions, namely "general races" and "data races". However, the formalism in Netzer *et al.* [70] is too simple to characterize the recently proposed complex race bug detection techniques such as multi-data block race bug detection techniques. Raza [81] surveys a dozen data race detection techniques, all of which belong to the class of data race bug detection techniques (see Section 2.3.3). Schimmel *et al.* [85] presents an empirical evaluation of bug detection capabilities of two data race bug detection tools on real-world concurrent programs. Wang *et al.* [111] and Qadeer *et al.* [80] present formal specifications of correctness criteria to detect race bugs. Violations of these correctness criteria correspond to the multi-block race bugs (see Section 2.3.6). Flanagan and Freund present a dynamic analysis framework, RoadRunner [34], that serves as a common platform to facilitate implementation of various race bug detection techniques for Java programs. They demonstrate that 11 race bug detection techniques could be implemented using RoadRunner.

Compared to those surveys, this paper systematically characterizes and classifies various race bug detection techniques of broader ranges (i.e., 43 race bug detection techniques) by using a formal execution model. I have presented formal definitions of the four classes of race bugs and analyzed the 43 race bug detection techniques according to their target race bug definitions. In addition, this survey analyzes and demonstrates the relations among race bugs of different classes (see Section 2.3.7). Furthermore, this survey reviews recently developed techniques such as multi-data race bug detection techniques.

There exist several survey papers on concurrent program analysis topics other than race bug detections for multithreaded programs. Tchamgoue *et al.* [99] surveys techniques for bug detection and debugging in event-driven programs and summarizes the terminologies in the techniques. Helmbold *et al.* [40] summarizes the concepts of the race bug detection techniques for parallel programs and presents a classification with respect to the characteristics of target program structure (e.g., loop, synchronization operations). Souza *et al.* [95] gives an overview of more than 50 papers on concurrent program analysis techniques. However, the paper includes only four papers on static/dynamic race bug detection techniques. Fiedor *et al.* [30] describes various concurrency errors including data races, atomicity violations, deadlock, livelocks, etc. and the corresponding bug detection techniques. However, the paper mentions only a few techniques briefly (less than 10 techniques summarized in one page) and does not analyze race bugs and detection techniques in detail. Lu *et al.* [58] reports a survey on the concurrency bugs from real software projects such as bug patterns, bug triggering conditions, and bug fix strategies. However, the paper does not analyze concurrency bug detection techniques. Bradbury *et al.* [10] compiles a set of Java concurrency bug patterns from the literature. Long *et al.* [55] presents a classification of Java concurrency bugs by using a Petri-net diagram. In contrast to the aforementioned works, this survey concentrates on the various race bug detection techniques for multithreaded programs.

2.4 Test generation techniques for multithreaded programs

Researchers have proposed automated testing techniques for multithreaded programs which can be categorized into three kinds of techniques: random noise injection-based test generation technique, bug-directed test generation techniques, and systematic test generation techniques. In this section, I describe each of the three kinds of multithreaded test generation techniques, and discuss the current limitation in generating effective and efficient tests for real-world multithreaded programs. In addition, I will discuss

the existing approach for the coverage-based test generation.

In this section, I do not cover the input data generation techniques for multithreaded programs (e.g. jCUTE [89]) since only few such techniques exist and test input generation is orthogonal to thread schedule generation.

2.4.1 Random noise injection-based testing techniques

Random noise injection-based test generation techniques insert artificial noises in a target program code to perturb thread schedules in executions. As a noise, some techniques inject sleep operations of random periods of time, or context-switching operations (i.e., `yield()`). In addition, a noise can be set to occur in a certain probability, in order to increase diversity in noise. These techniques insert noise-injecting probes before/after an operation whose execution order affects the target program behavior. (i.e., synchronization operation, or shared variable accesses)

ConTest [27] is a multithreaded Java testing framework based on noise injection techniques. ConTest instruments a target program for inserting random sleeping as noise probes before/after synchronization and shared variable access operations. In addition, ConTest utilizes static analyses to predict concurrency bugs, and then insert probes at the predicted bugs to produce error-inducing thread schedules efficiently. Moreover, ConTest also utilizes the coverage feedback for determining effectiveness noise configurations at test generation [51, 102]. Rstest [96] is another noise injection based technique for multithreaded Java programs. Rstest uses yield operation as noise probe instead of sleep to reduce unnecessary slowdown in execution time when artificial sleep noises are used.

Random noise injection based testing techniques indirectly induce a target program to generate various concurrent behaviors, rather than controlling thread schedules explicitly. Thus, they can be used for detecting any type of concurrency errors. As these techniques do not use sophisticated analyses on the target programs, random noise injection-based techniques can be used for testing of multithreaded Java programs with complicated synchronization (e.g., ad-hoc synchronization mechanism), or the programs where only partial code/binary is available.

However, random noise injection-based techniques are not effective for detecting corner-case concurrency errors and they are ineffective for concurrency error detections in general. Due to random nature of thread schedule generation, there exists no guarantee that they will explore subtle thread schedules progressively. As a consequence, these techniques may be not effective for detecting concurrency errors that hide under subtle thread schedules, although test generation continues for a large amount of time. Second, injected noises incurs significant runtime overhead in test executions, which degrade testing efficiency. Last, the performance for testing an arbitrary target program depends largely on noise injection configuration (e.g., noise type, randomness parameters); however, there are no method to find best configurations before test generation starts.

2.4.2 Bug-directed testing techniques

The active testing techniques generate test executions by manipulating the execution orders of specific operations in runtime of test executions. These techniques aim to generate specific thread schedules that seem useful for discovering concurrency errors. An active testing technique selects the targeted operations based on concurrency bug prediction results or coverage metrics before test generation. In the test generation phase, an active testing technique generates the specific execution orders of the selected operations, that are expected to induce targeted concurrency behaviors.

CalFuzzer [48] is a bug-directed active testing technique that utilizes concurrency bugs predicted by a data race detector [88], an atomicity violation detector [75], and a deadlock detector [49] as test targets. For each type of concurrency bugs, CalFuzzer uses a specific thread schedule algorithm to generate error-inducing thread schedules. PECAN [46] is another bug-directed active testing technique which targets more kinds of concurrency bugs than CalFuzzer.

The main challenge for the bug-directed testing technique is the quality of generated test targets. The fault detection capability of an active testing technique depends on the kinds of test targets. Unless a technique selects the operations related to a concurrency error, the technique is not effective to generate the thread schedules related to the concurrency error. For this reason, the bug-directed active testing techniques are inherently not capable of detecting general concurrency errors which do not correspond to any predefined patterns.

2.4.3 Systematic testing techniques

The systematic test generation techniques for multithreaded programs generate thread schedules by manipulating all execution orders of operations in a test execution. These techniques intercept every execution of the operations related to concurrent behavior of a target programs to control exact orders of their executions. As a result, these techniques generate a finite sequence of concurrency operations for a test execution. If a test explores all possible operation sequences for a multithreaded program with a given input data, the test can detect all existing concurrency errors, or guarantee that there is no error. For this reason, the systematic test generation techniques are often called as stateless model checkers as these techniques do not store visited states.

A challenge in the systematic test generation techniques is that the number of possible operation sequences grows exponentially with respect to the multithreaded program size, and thus, for most real-world programs, there are an enormously large number of possible sequences. To resolve this challenge, a systematic test generation technique has a search strategy that aim to generate useful operation sequences for detecting concurrency errors cost-effectively. In high-level, there are three types of search strategies: reduction, selection, and prioritization.

- **Reduction strategies.** The techniques with the reduction strategies check whether or not two operation sequences are semantically equivalent, and avoid generating equivalent sequences since their testing results are redundant. RAPOS [87] is a systematic test generation technique that generates random operation sequences while its search strategy leverages a dynamic partial order reduction algorithm to reduce useless effort for randomizing independent concurrent operations.

- **Selection strategies.** The selection strategies lead generated thread schedules to satisfy a certain condition which provides usefulness of the generated thread schedules. The techniques with the selection strategies first define the targeting thread schedule conditions before test generation. And then, the techniques generate the thread schedule decision if it makes the execution satisfy the predefined condition. Inspect [110] uses a concurrency bug detector such as data race detector to identify concurrency bugs, and then the search strategy leads an execution to cover an error-inducing thread schedule of a predicted concurrency bug. Fusion [109] uses the HaPSet coverage metrics to create the targeted conditions, and then generates each thread schedule to cover new HaPSet test requirements. One drawback is that the tests generated with the search strategy are not able to detect the concurrency errors if the errors do not correspond to the predefined target

condition. Therefore, the techniques with the selection strategies are useful only for detecting concurrency errors of specific types.

- **Prioritization strategies.** A prioritization strategy arranges thread schedules with respect to certain criteria, and then generate test executions according to the thread schedule in the order. Most prioritization criteria assign a higher priority to a simpler thread schedule, such that a generated test first explores comprehensive executions under simple thread schedules, and then gradually explores more executions with complicated thread schedules. The intuition behind these strategies is that many concurrency errors can be detected with simple thread schedules. Thus, it is cost-effective to explore check for the comprehensive cases of simple thread schedules before complicated thread schedules. The context-bounded search strategy counts the number of context-switches in an operation sequence, and gives a higher priority to an operation sequence with a less number of context-switches [65]. PCT leverages the context-bounded search strategy with randomness to provide a guarantee to detect concurrency errors of a certain complexity with a certain probability [12]. The delay-bounding search strategy measures the difference between an operation sequence and the operation sequence with a round-robin scheduling, and give a higher priority to a sequence with less difference [28]. The empirical studies imply that these search strategies are effective because most concurrency errors are detected with simple thread schedules.

Despite the efforts to cover the thread schedule space of a target program efficiently, the systematic test generation techniques to date are not not effective with large target programs at detecting concurrency errors. Although they can detect all existing concurrency errors with unlimited testing time, these techniques can only explore limited space of thread schedule within realistic bound of time and memory resources. Furthermore, the runtime overhead for generating each test execution is large, thus these techniques would not be cost-effective for detecting concurrency errors in a large-scale multithreaded program.

2.4.4 Coverage-based test generation

Maple [120] uses a concurrency coverage metric *iRoot* which generates a test requirement consists of two to four instructions. Maple utilizes a dynamic analyzer to obtain the target test requirements which are expected to be achievable. For each test execution, Maple chooses one of the uncovered test requirements per test generation, and controls the execution order of the operations that are related to the chosen test requirement to satisfy the test requirement.

For the coverage-based testing techniques, the quality of test targets depends on the adequacy of the target coverage metric, and the method to obtain achievable test requirements. The iRoot metric used by Maple generates test requirements to capture one or two thread interactions between two threads. Maple predicts achievable iRoot test requirements of a target program based on the dynamic analysis result prior to a test generation. However, there is no evidence that the iRoot metric generates sufficient test requirements to suitable for detecting general concurrency errors. Moreover, there is no guarantee that all achievable test requirements are predicted by the dynamic analysis.

Maple is the only coverage-based test generation to the best of my knowledge, and most closely related to the coverage-based testing techniques to be presented in a later chapter. More comparisons of my technique with Maple will be found in Section 4.6.3.

Chapter 3. Empirical Evaluation of Concurrency Coverage Metrics

3.1 Introduction [1]

Concurrent coverage criteria define a set of test requirements for a multithreaded program, which enumerates a set of possible interleavings of synchronization operations or shared variable accesses. As similar to branch and statement coverage criteria, concurrent coverage criteria aim to allow testers to estimate how well they have exercised concurrent program behaviors.

The intuition behind all concurrent coverage criteria is that as more test requirements relative to the criteria are satisfied, the testing process is more likely to detect faults. Thus, to maximize the effectiveness of testing processes, researchers create test adequacy criteria based on these metrics, and develop techniques to satisfy them. The development of such techniques has long been an active area of research in the context of structural coverage metrics for single threaded programs [14, 37, 74, 101], and as multithreaded programs have become more common the development of techniques centered around concurrent coverage criteria has also become an active area of research [27, 41, 51, 109].

Unfortunately, the intuition behind concurrent coverage criteria remains largely unexplored. While a large body of evidence exists exploring the impact of the coverage criteria invented for single threaded programs on testing effectiveness (e.g., [4, 69, 124]), we are aware of no study rigorously examining the impact of concurrent coverage criteria. We expect that increasing coverage relative to these criteria will improve testing effectiveness, but we also expect that it will increase test suite size. Thus we must ask: does improving concurrent coverage directly lead to a more effective testing process, or is it merely a byproduct of increasing test suite size? Further, if improving coverage does lead to increased testing effectiveness what practical gains in testing effectiveness can we expect? Finally, based on the effectiveness of the current state of the art concurrent coverage metrics, what steps should be taken with respect to continuing the development of test case generation techniques for concurrent coverage criteria?

In researches on concurrent coverage criteria, the effectiveness of achieving high coverage has been argued for primarily through analytical comparisons between coverage definitions and concurrency fault pattern, such as those involving data races and atomicity violations [56,102,109] (see Section 2.2.3). Our study's scope is more comprehensive, encompassing twelve case examples and eight concurrent coverage criteria, and we apply a broader set of analyses.

To explore these questions, we studied the application of eight concurrent coverage criteria (see Section 3.2) in testing twelve concurrent programs. For each program and metric pair, we used a randomized test case generation process to generate 90,000 test suites with varying levels of size and coverage, and measured the relationships between the percentage of test requirements satisfied, the number of test executions, and the fault detection ability of test suites via correlation and linear regression. Additionally, we compared test suites generated to achieve high coverage against random test suites of equal size.

Our results show that each coverage criterion explored has value in predicting concurrency testing effectiveness and as a target for test case generation. However, the results of the metrics vary across

[1] Parts of this chapter were presented in ICST 2013 [43] and published in the STVR journal [44].

programs, which is contrast to the characteristics of the coverage metrics for single threaded program testing [69]. In particular, we found that the correlation between concurrent coverage and fault detection, while often moderate to strong (i.e., 0.4 to 0.8) and stronger than the relationship between test suite size and fault detection, is occasionally low to non-existent.

We also found that while large increases in fault detection effectiveness (up to 25 times) can be found when using concurrent coverage criteria as targets for test case generation relative to random test suites of equal size, in some cases the results were no better than random testing.

Given these results, we see that the proposed concurrency coverage criteria have value and efforts to develop techniques based on these coverage criteria are justified;however, additional work on more advanced concurrency coverage criteria is required. In particular, the variability in metric effectiveness across programs highlights the need for guidelines to help testers select from among the many metrics already proposed. The coverage criteria would be improved to better capture the factors that constitute effective concurrency testing.

3.2 Study design

The purpose of this study is to rigorously investigate the concurrency coverage metrics presented in previous work, and to either provide evidence of each metric's usefulness or demonstrate that the metric is of little value. The usefulness of a coverage metric, concurrency or otherwise, invariably relates to many factors, such as the testing budget available, the characteristics of the program under test, and the goals of the testing process. Nevertheless, to show that any coverage metric can be considered useful, it is necessary at minimum demonstrate two things:

- increased levels of coverage correspond to increased fault detection effectiveness;
- these increases are due in part to increasing coverage levels, not merely larger test suite sizes.

Further, to aide practitioners in selecting a coverage metric for use, we should attempt to quantify the relationship between coverage, size, and fault detection effectiveness. In particular, we are interested in the predictive value of each metric and the expected improvements over random testing in terms of fault detection.

Finally, we are interested in how, given the concurrency coverage metrics proposed, we can best approach test case generation for concurrent systems. Specifically, we wish to know whether potential issues with these metrics, already identified in our previous work [43], can be overcome by a combined use of coverage metrics. We also wish to know whether the current state of the art, coverage-guided test generation techniques for concurrent program testing could be improved by the development of techniques targeting difficult-to-cover test requirements. Such techniques would be analogous to existing methods for improving coverage when using sequential coverage metrics, for example symbolic execution and genetic algorithm based approaches [37, 114].

Our study is thus designed to address four core questions.

- **Research Question 1 (RQ1):** *For each concurrency coverage metric studied, does the coverage achieved positively impact the effectiveness of the testing process for reasons other than increases in test suite size?* In other words, we would like to provide evidence that given two test suites of equal size, the test suite with higher coverage will generally be more effective.

- **Research Question 2 (RQ2):** *For each concurrency coverage metric studied, how does the fault detection effectiveness of test suites achieving maximum coverage compare to that of random test suites of equal size?* While coverage levels may relate to effectiveness, the practical impact of achieving high coverage for some metric over random test suites may be insignificant.

- **Research Question 3 (RQ3):** *For the concurrency coverage metrics studied, do combinations of coverage metrics outperform the original coverage metrics?* The effectiveness of coverage metrics can vary, with the most effective metric varying from case example to case example. By combining metrics, we can potentially overcome these inconsistencies.

- **Research Question 4 (RQ4):** *For each concurrency coverage metric studied, does covering difficult-to-cover test requirements result in above average fault detection relative to other coverage requirements?* For a given case example, some coverage metrics contain test requirements that are hard to cover, i.e. a small percentage of possible test cases satisfy the requirement, and thus achieving maximum coverage in such scenarios can require significant effort. We would like to determine whether such effort is potentially justified.

The objects for this study have been drawn from existing work on concurrent software analysis [26, 71, 75], and include objects without faults, and objects with faults detected in previous studies. Each object is a multithreaded Java program.

We list the objects with the lines of code, numbers of threads, the type of test oracle for the program, and mutants used in Table 4.1. The *LOC* column represents the size of the original source code for each subject. The *number of threads* column shows how many threads are created during test execution, as determined by the test case given for each object. The *test oracle* column describes the test oracle used for the program. "AS" means that the fault is detected by an assertion that checks application-specific requirement properties, and the number in the parenthesis represents the number of assertion statements in the program. "TO" means that the fault is detected by a timeout (i.e., deadlock). The *incorrect versions* column represents, for the mutation testing objects, the number of generated mutants and the number of mutants used in parenthesis (the reason for the differences in these numbers is explained in Section 3.2.2).

3.2.1 Variables and measures

Independent variables.

In this study, we manipulate two independent variables: the concurrency coverage metric and the method of test suite construction.

Concurrency Coverage Metrics. Numerous concurrency coverage metrics have been proposed, each based on some intuition about how to capture different aspects of concurrent executions. We view these metrics as having two key properties: the *number of code elements* the test requirements consider (either a single element or a pair of elements), and the the *code construct* the metric is defined over (either synchronization operations or data access operations). For example, the *Blocking* and *Blocked* coverage metrics define test requirements based on individual `synchronized` blocks/methods in a Java program [27], and are thus *singular* concurrency coverage metrics, while the *Blocked-Pair* metric is defined over pairs of blocks, and is thus a *pairwise* metric. All of these metrics are defined over `synchronized` blocks, and thus they are all synchronization metrics [102].

Table 3.1: Study objects used for the empirical study on concurrency coverage metrics

Type	Program	LOC	Number of threads	Test oracle	Incorrect versions	Number of test executions
Mutation testing	ArrayList	5866	29	AS(6), TO	42 (10)	2000
	Boundedbuffer	1437	31	AS(6), TO	34 (6)	2000
	Vector	709	51	AS(15), TO	88 (35)	2000
Single fault program	Accountsubtype	193	12	AS(1)	1	1000
	Alarmclock	125	4	AS(1)	1	1000
	Clean	51	3	TO	1	1000
	Groovy	433	3	TO	1	1000
	Piper	71	9	TO	1	1000
	Producerconsumer	87	5	AS(1)	1	1000
	Stringbuffer	416	3	AS(19)	1	1000
	Twostage	52	3	AS(1)	1	1000
	Wronglock	118	22	TO	1	1000

Table 3.2: Concurrency coverage metrics used in the study

	Synchronization operation	Data access operation
Singular	Blocking [27], Blocked [27]	LR-Def [56]
Pairwise	Blocked-Pair [102], Follows [102], Sync-Pair [41]	PSet [119], Def-Use [98]
Combined	Blocked-Pair+Def-Use, Follows+Def-Use, Sync-Pair+Def-Use,	Blocked-Pair+PSet Follows+PSet Sync-Pair+PSet

We selected eight coverage metrics for use in our study, focusing on well-known metrics while also ensuring that we considered every possible combination of our two key properties. We list the metrics selected in Table 3.2. We concentrated on metrics that generate modest numbers of test requirements, as this makes achieving high levels of coverage feasible in a reasonable time. Thus, coverage metrics that produce very large numbers of test requirements are not included in this study. These include metrics defined over memory addresses or exhaustive sets of interleavings (e.g., *all-du-path* [115], *ALL*, *SVAR* [56]) and the series of extended coverage metrics proposed by Sherman *et al.* [92]. *Access-pair* [92] and *location-pair* [98] are omitted as they are almost equivalent to the *PSet* metric. We interpret the *LR-Def* metric as generating two test requirements for read accesses: one for the use of memory defined by a local thread and the other for the use of memory defined by any remote thread.

In addition to these metrics, we considered six coverage metrics that are combinations of these metrics to investigate the benefits of combining existing metrics (to address *RQ3*). Each combined metric was created by combining the test requirements of one pairwise synchronization based coverage metric (i.e., *Blocked-Pair*, *Follows*, and *Sync-Pair*) and the test requirements of one pairwise data access based coverage metrics (i.e., *Def-Use*, and *PSet*). Hereafter we refer to the non-combined metrics as *original coverage metrics*, and the six new coverage metrics as *combined coverage metrics*.

We chose these combinations for three reasons: (1) synchronization based coverage metrics and

data access based coverage metrics represent different paradigms for measuring concurrency coverage, and thus seem likely to be complementary; (2) metrics within a paradigm tend to achieve similar coverage and fault detection effectiveness rates; and (3), pairwise metrics generally outperform singular metrics (at least as test case generation targets), and thus make a better starting point when attempting to improve concurrency coverage metrics.

Test Suite Construction. We used two methods of test suite construction: random selection and greedy test suite reduction. In random selection, test suites are constructed by randomly selecting test executions to construct test suites of specified sizes. In greedy selection, test suites are constructed to achieve maximum achievable coverage using a small number of test executions. These test suite construction methods are used to address *RQ1* and *RQ2*, respectively.

Dependent Variables

We measure three dependent variables computed over generated test suites: coverage achieved, test suite size, and fault detection effectiveness. Additionally, we measure two dependent variables computed over test requirements: difficulty of covering test requirements, and the fault detection effectiveness achieved when covering test requirements.

Achieved concurrency coverage of test suites. For a give metric M, each test suite S's coverage is computed as the ratio of M's test requirements that are satisfied by S to the total number of test requirements satisfied across *all* executions for a given program *version*. We construct test executions while holding random test case generation parameters constant (see Section 3.2.2); because different parameters can result in covering different requirements, the coverage of M's requirements is often less than 100%, and our measurements reflect this. However, for the purpose of greedy test suite construction, we define *maximum achievable coverage* as the number of requirements than can be covered for a specific set of test case generation parameters.

Test suite size. Test suite size is the number of test cases in the test suite, and estimates testing cost.

Fault detection effectiveness of generated test suites. The fault detection effectiveness of a test suite is "success" when the fault is detected by at least one execution of a test case in the test suite , or "failure" when the fault is not detected by any test case execution. During analysis we typically compute the average fault detection effectiveness across many test suites, with results that range from 0.0 to 1.0.

Difficulty of satisfying test requirements. The difficulty of satisfying each test requirement is computed as the ratio of the number of test executions satisfying the requirement to the total number of test executions.

Fault detection effectiveness of test requirements. The fault detection effectiveness of a test requirement is the ratio of the number of test executions detecting a fault while covering the test requirement to the number of test executions that cover the test requirement.

3.2.2 Experiment setup

Conducting our experiment requires us to:

1. generate mutants for programs without faults,
2. conduct a large number of random test executions,

Table 3.3: Mutation operators

Category	Description
Change Synchronization Operations	Exchange Synchronized Block Parameter
	Remove `wait()`
	Replace `notifyAll()` with `notify()`
Modify Synchronized Block	Expand Synchronized Block
	Remove Synchronized Block
	Remove `synchronized` Keyword from Method
	Shift Synchronized Block
	Shrink Synchronized Block
	Split Synchronized Block

3. for each execution, record the requirements covered for all metrics and whether a fault is detected,
4. compute the difficulty and fault detection rate for each requirement generated,
5. perform resampling over executions to construct test suites, and
6. measure the resulting coverage and fault detection effectiveness of each test suite.

Mutant Generation

We wished to study fault detection in the presence of many diverse fault types, which is not possible when using single fault programs. Thus, for several of our object programs we corrected known faults [75] and applied mutation analysis. To choose mutation operators for our study, we drew on concurrency mutation operators used in a recent survey on concurrency mutation testing [9]. These operators are similar to traditional syntactic mutation operators commonly used in other studies [4, 25], but focus on manipulating synchronization constructs, e.g., adding and removing synchronization primitives. Table 4.2 describes the operators. We applied these operators to generate mutants. We then discarded any mutants that (1) did not fail for any generated test execution, (2) were malformed, e.g., resulted in code that could not be executed, or (3) were killed by every test execution.

We list the number of mutants generated together with the final number of mutants used within parentheses in Table 4.1. Note that we also use objects containing real faults, thus mitigating the risk present when using concurrency mutation operators, whose usage is less established and studied than structural mutation operators for sequential programs [4]. Hereafter, when referring to "objects" we are referring to individual faulty programs, e.g. "all objects" refers to all single fault programs and all mutants.

Test Generation and Execution

We used a randomized test case generation approach to avoid bias that might result from using a directed test generation approach such as those proposed in [27,96]. Our approach selects an arbitrary test input and generates a large number of test executions by executing a target program on the test input with varying random delays (i.e. calls to `sleep()`) inserted at shared resource accesses and synchronization operations.

We control two parameters of this approach: the *probability* that a delay will be inserted at each shared resource access or synchronization operation (0.1, 0.2, 0.3, and 0.4), and the maximum length of

the *delay* to be inserted (5 milliseconds, 10 milliseconds, and 15 milliseconds). We used these controls because prior work indicates that they can impact the effectiveness of the testing process [51]. The specific values used were selected based on our previous experience in this domain [41] and pilot studies, both of which indicated that larger or finer grained delays and probabilities did not yield significantly different results. In addition to the twelve random scheduling techniques, we ran test executions without inserting any delay noise.

We began by estimating the number of test executions E required to achieve maximum coverage for all eight coverage metrics used, and each of the six combined metrics considered. This was done by executing the original object for several hours and recording the rate of coverage increase for each metric. For each object, we required either 1000 or 2000 test executions. Following this, for each parameter setting (13 (=4×3+1) in total) we conducted E executions for each mutant (for objects with mutants) or each object program (for objects without mutants). During each execution, we recorded (1) the test requirements covered for each coverage metric studied, and (2) whether a fault was detected. We recorded an execution as detecting a fault if (1) an application-specific assertion statement is not satisfied (i.e., invariant violations), (2) a crash occurs that throws a uncaught exception (e.g., null pointer dereference, array index out-of-bound, invalid memory access), or (3) the program deadlocks, determined by checking whether execution time is exceptionally long.

Data Collection

After each test execution we know (1) which test requirements are covered for each coverage metric and (2) whether the program failed. Based on this information, we can obtain the data for each test requirement – how frequently the test requirement is covered and how frequently executions that cover the test requirement detect a fault. This data is used for analysis related to *RQ4*.

Using the test execution information, we can, via random resampling, construct test suites of varying sizes and levels of coverage. Ideally, we would like to construct test suites encompassing all possible combinations of size and coverage. Unfortunately, as coverage and size tend to be highly correlated this is impossible; small test suites with high coverage (or vice-versa) are extremely rare in practice. We instead generated, for each combination of object and coverage metric, 90,000 test suites ranging in size (i.e. number of test executions) from 1 to the maximum size via random sampling of executions. This results in a set of test suites with increasing size and, within each level of size, varying coverage. These test suites are used to help address *RQ1*, *RQ2* and *RQ3*.

We also generated 100 test suites achieving maximum achievable coverage for each coverage metric. We generated these using a *mostly* greedy test suite reduction approach: from the set of executions, repeatedly select either (1) the test execution satisfying the most unsatisfied requirements (80% chance) or (2) a random test execution (20% chance) until all requirements are satisfied. This results in a test suite that achieves maximum coverage using fewer test executions than are required by simple random test suite construction. The randomization adds noise, ensuring some variation in the generated suites. These test suites are used to address *RQ2*. To investigate *RQ3*, we apply the same test construction for the six combined coverage metrics as well.

To select a test suite for a single fault program or mutant, we have one set of executions over the object, and we resample from this set to construct test suites. Each test suite becomes a data point for analysis, having an associated level of coverage, size, and fault detection result (killed/not killed). When constructing each test suite, we held probability and delay constant. This was done to facilitate later analysis considering the impact of these factors.

Note that the generation process for the original eight metrics and the six combined metrics is the same. We treat a combined metric (e.g. *Follows+PSet*) as a single metric, with its own separate set of coverage requirements, a separate sets of greedy test suites, etc. This allows for a fair comparison of the original and combined metrics in Section 3.3.

3.2.3 Threats to validity

External: We conducted our study using only Java programs with standard synchronization operations. These programs are relatively small but have been chosen from existing work in this area, and thus we believe that our results are at least generalizable to the class of programs concurrent program testing research focuses on.

For concurrency coverage metrics, it is difficult to accurately determine satisfiable test requirements. For all coverage metrics, however, we appear to have reached saturation during test case generation (see Section 3.3.1) [92], and thus a larger number of executions is unlikely to significantly alter our results.

The randomized test generation technique we use was implemented in-house, but we have attempted to match the behavior of other random testing techniques by constructing a general technique and varying the parameters of probability and delay. We follow the current practice of concurrency testing research which focuses on analyzing diverse thread interleavings effectively and efficiently by restricting other factors such as test inputs. Thus, our study utilizes various thread schedulings with single test input values, which may not consider the impact of various test input values on concurrent program testing.

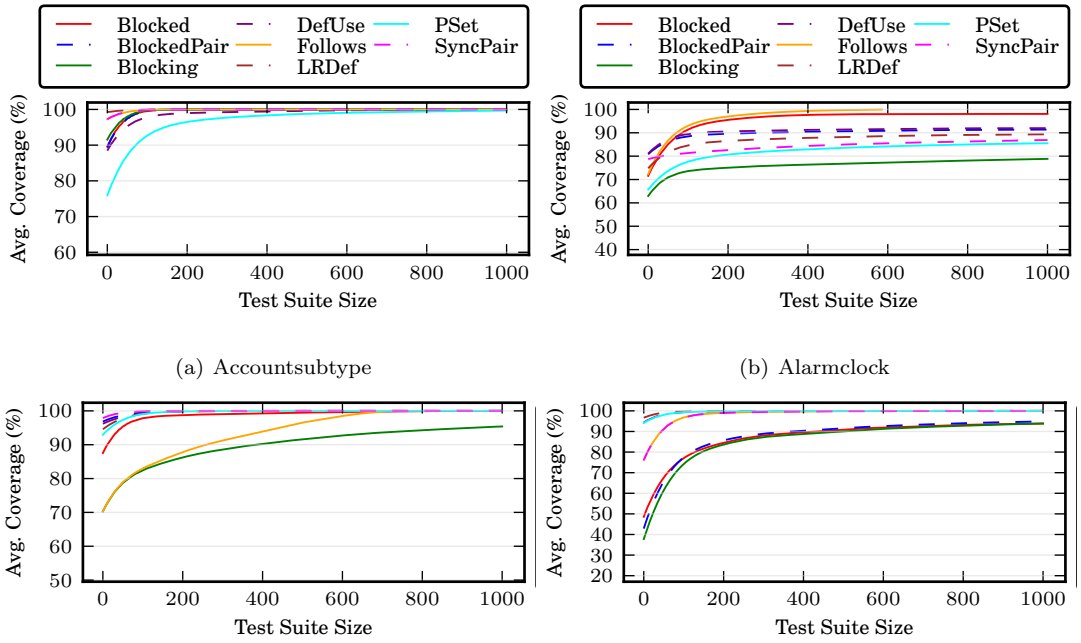
Internal: Our randomized test case generation technique is implemented on top of Java's internal thread scheduler. When using other algorithmic thread schedulers, such as PCT [12,67] or CTrigger [76], results may vary. Additionally, while we have extensively tested our experimentation tools, it is possible that faults in our tools could lead to incorrect conclusions.

Construct: Our method of detecting faults may miss faults, e.g., errors not captured by an assertion violation or not leading to an exception. In practice, however, much of concurrent testing focuses on detecting faults via imperfect test oracles and thus our study uses a realistic approach to fault detection.

We measured the maximum coverage for a metric by tracking all coverage requirements covered in any execution during test generation. This value is likely lower than the actual maximum achievable coverage because there likely exist coverage requirements that are achievable but not covered by any generated execution. Nonetheless, since we generate a large number of executions with different random testing techniques, we expect missed coverage requirements are few. Furthermore, even if the maximum coverage values are incorrect, only *RQ3* depends on this value and thus other conclusions drawn would not change. We did not use a predictive analysis technique for the study because the existing predictive analysis techniques are known to produce false positives (i.e., infeasibile test requirements are estimated as feasible).

We used mutation analysis to measure testing effectiveness for some objects. Our seeded faults are designed to mimic actual concurrency faults, and of course are indeed faults, but the relationship between faults generated by concurrency mutation operators and real concurrency faults has not been thoroughly investigated. Nevertheless, the results for mutation-based objects and objects with real faults are similar.

Conclusion: For each object, we constructed from 1 to 88 faults and 100,000 test suites per coverage metric. While more mutants, faults, and test suites could in theory alter our conclusions, in practice our conclusions remain the same for both single fault programs, mutation-testing driven programs, and larger numbers of test suites.



(a) Accountsubtype

(b) Alarmclock

(c) Clean

(d) Groovy

Figure 3.1: Size versus coverage, four single fault objects

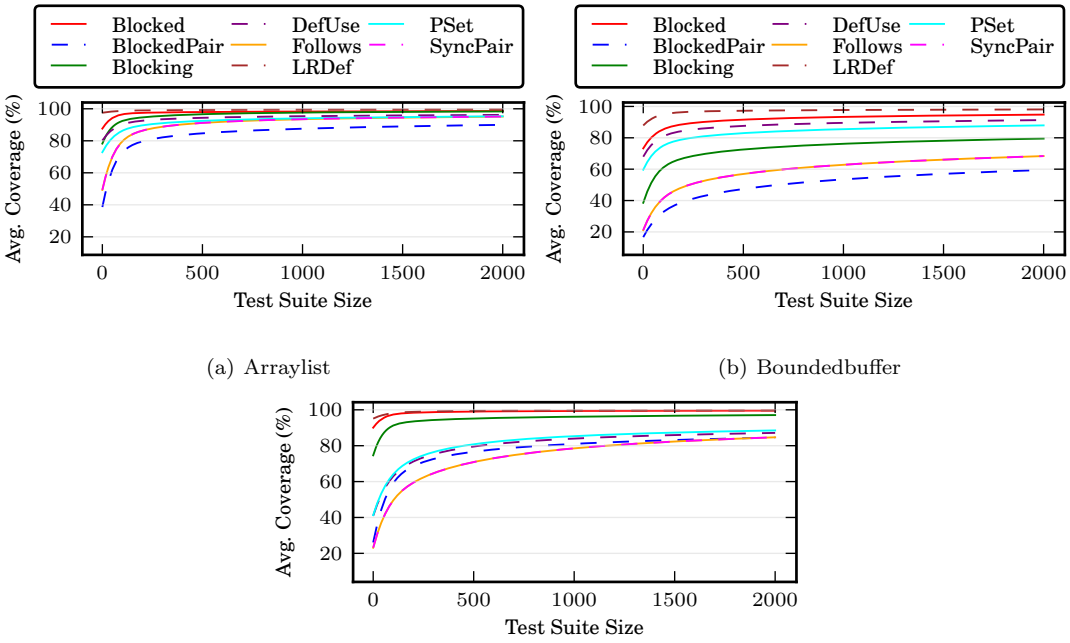
3.3 Results

Our analyses are designed to study how each coverage metric impacts fault detection effectiveness. Towards *RQ1*, we visualized the pairwise relationship between variables, measured the correlation between coverage, size, and fault detection effectiveness, and performed linear regression to better understand how both coverage and size contribute to fault detection effectiveness. Towards *RQ2*, we compared the fault detection effectiveness of test suites satisfying maximum achievable coverage and random test suites of equal size. Towards *RQ3*, we performed the analysis above over combinations of pairwise metrics and compared the results with the single metric versions. Finally, towards *RQ4* we examined the correlation between the difficulty of covering a test requirement and the average fault detection for test executions covering a test requirement, and compared the average fault detection for difficult-to-cover to the fault detection for easy-to-cover test requirements.

Ideally, we would like a coverage metric that: (1) is highly correlated with fault detection (over 0.7 coverage); (2) along with size, results in regression models with high fit for fault detection (higher than 0.8); and (3) allows us to select test suites with significantly higher fault detection than randomly selected test suites of equal size (improvements in fault detection of at least 20%). Any metric fitting such criteria would be useful both as a predictor of fault detection effectiveness and as a test generation target.

3.3.1 Visualization

To understand the relationship between test suite size, coverage, and fault detection effectiveness, we began by plotting the relationship between each pair of variables. In Figure 3.1 we show the relationship between size and coverage for each coverage metric, for four single fault objects (Figure B.1 for all single fault objects). In Figure 3.2 we show the same relationship for objects using mutation testing. In Figure 3.3 we show the relationship between coverage and fault detection for four single fault objects



(a) Arraylist

(b) Boundedbuffer

(c) Vector

Figure 3.2: Size versus coverage, mutation objects

(Figure B.2 for all single fault objects). In Figure 3.4 we show the same relationship for objects using mutation testing. Finally, in Figure 3.5 we show the relationship between size and fault detection for all objects. Note that expanded versions of Figure 3.1 and 3.3 are found in Appendices B. To ease readability, we have elected to show only specifically referenced objects here.

Recall from Section 3.2.2 that for each combination of probability and delay (two variables controlled during test generation) 1000 test executions were generated for each single fault program. Each figure is an average across these traces of the test executions. Additionally, rather than plot a separate figure for each of the dozens of mutants for the *Arraylist*, *Boundedbuffer* and *Vector* objects, figures for these objects are averages across all mutants. Note that this averaging results in figures that do not necessarily reflect the underlying trends within each mutant, as we discuss later in this section.

In all of the figures, there is typically a fair amount of variation along the y-axis as coverage and size increase. To improve the readability of the figures, we have used two forms of smoothing. In the case of plots of size versus coverage and size versus fault detection, we have used LOESS smoothing with a factor of 0.1. The relationships here are clearly visible with raw plots; the use of mild smoothing allows us to distinguish coverage metrics and objects after plotting. However, plots of coverage versus fault detection are very noisy, as indicated by the correlations shown in Section 3.3.2. LOESS smoothing is of limited help here, and so to further improve readability, before plotting we have averaged the fault detection rates for all coverage levels within 5 percentage points, i.e. we have averaged the fault detection rate for test suites achieving 12.5-17.5% coverage, 17.5-22.5% coverage, etc.

This averaging across mutants and test generation parameters results in graphs that must be carefully interpreted: individual points on the lines can reflect the average of many test suites — particularly for coverage levels above 50% — or few test suites, as very low coverage levels are infrequently achieved in practice. This is unfortunate, but necessary, as the alternative is to either plot each combination of coverage metric and object separately, which would require hundreds of figures, or as very dense scatter-

plots, resulting in unintelligible figures. However, the goal of visualization is just to spot broad trends; rigorous analysis follows in the remaining sections.

Note that in several cases coverage achieved is less than 100%. This occurs because each test suite is specific to a single combination of test generation parameters, but the set of test requirements (and thus the mark for 100% achievable coverage) is computed across all test suites. Thus, it is possible that no single test suite achieves 100% maximum achievable coverage. Similar behavior is shown in Figure 3.5, as several test suites of maximum size fail to detect the fault.

We begin by examining the relationship between size and coverage/fault detection, as shown in Figures 3.1, 3.2 and 3.5). We can see that the concurrency coverage metrics often — but clearly not always — exhibit behavior similar to what we expect from sequential coverage metrics and testing: broadly logarithmic behavior, with a rapid increase in both fault detection and coverage for small test suite sizes, and smaller increases as test suite size increases. Here we see small differences in coverage metrics: some coverage metrics begin with very high levels of coverage for even small test suites, and thus quickly achieve close to maximum coverage, while others grow in coverage more slowly. For example, *LR-Def* is an extreme case, achieving maximum coverage almost immediately for many programs. In contrast *Follows*, a more complex metric, often achieves maximum coverage only with larger test suites sizes, i.e., those greater than 300. Here, differences are related primarily to the number of "easy" requirements to satisfy – those metrics that are easier to satisfy have high coverage even for very small test suites, e.g., *Blocking*, *Blocked*, *LR-Def*. Similar variations are also visible in the relationship between size and fault detection (see Figure 3.5). On the whole, however, the relationship between size and coverage/fault detection is clearly positive.

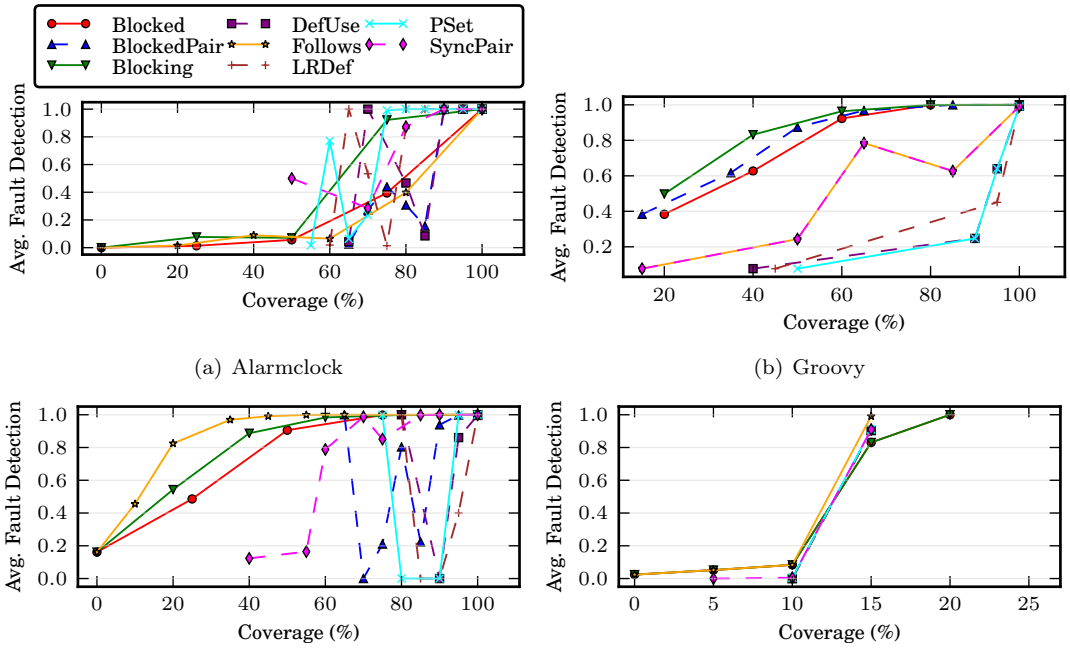
Less easily inferred from the figures is the relationship between coverage and fault detection (Figures 3.3 and 3.4). Clearly, in many cases the relationship is positive; for example, this is true for all metrics when applied to the *Twostage* and *Arraylist* objects. In other cases the relationship is noisy, but nevertheless, high coverage appears to result in high fault detection, for example on the *Alarmclock* object. In some cases, however, the relationship is quite unclear. *Boundedbuffer*, for example, exhibits no clear pattern for any coverage criteria (except when testing one specific mutant, as we discuss later), whereas *Blocked-Pair* coverage varies from seeming clearly related to fault detection (e.g., for the *Groovy* and *Vector* objects) to seeming marginally related to fault detection (e.g., for the *Alarmclock* and *Stringbuffer* objects).

This clear positive relationship between size and fault detection, coupled with the inconsistent, but nevertheless positive relationship between coverage and fault detection, provides informal evidence that both size and coverage impact fault detection effectiveness. We quantify the impact of both factors in the following subsections.

3.3.2 Correlation between variables

The foregoing visualizations indicate that both test suite size and coverage appear to be positively correlated with fault detection effectiveness, and that size is positively correlated with coverage. To measure the strength of these relationships, for each object and coverage metric we measured the correlation between each variable using Pearson's r.[2] We selected Pearson's r for two reasons. First, we are interested in the application of concurrency coverage metrics as predictors and thus measuring the strength of the linear relationship between variables is desirable. Fault detection is guaranteed to increase

[2]For small samples, conclusions based on Pearson's can be unsound for non-normal data; in our case the use of very large number of samples, 30,000-90,000 per correlation computed, mitigates this risk.



(a) Alarmclock (b) Groovy

(c) Stringbuffer (d) Twostage

Figure 3.3: Coverage versus fault detection effectiveness, four single fault objects

monotonically with size and coverage, and thus establishing this using rank correlation (e.g. Spearman or Kendall's tau) yields less new information [50]. Second, single fault programs can only fail or pass for each test suite; computing correlation over such data is a special case known as point-biserial correlation, for which rank correlation (due to the many ties present) is unsuitable. For every non-zero correlation computed, the p-value was (far) less than 0.05 and thus statistically significant at $\alpha = 0.05$.

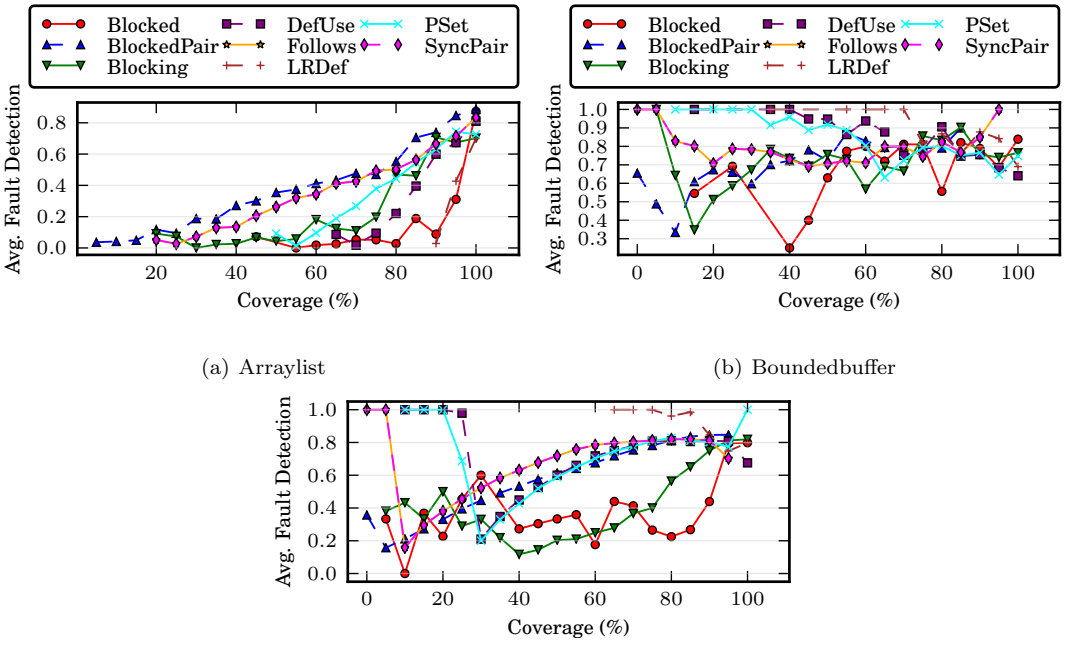
The computed correlations for single fault programs are presented in Table 3.4. For example, for *Accountsubtype*, the correlation between *Blocked* coverage and fault detection/test suite size is 0.39 and 0.11, respectively, while the correlation between size and fault detection (*S-FF*) is 0.22, indicating that coverage is more highly correlated with fault detection than test suite size.

The correlations for objects with multiple faulty versions are shown as boxplots in Figure 3.6.[3] The column labeled *X-FF* represents the correlation between the coverage X and fault detection, and the column with *X-SZ* represents the correlation between the coverage X and the test suite size. The last column labeled *S-FF* is the correlation between test suite size and fault detection.

For example, we can see for *Arraylist* that the correlation between size and fault detection (column labeled "S-FF") ranges from 0.4 to slightly less than 0.2, with a median slightly under 0.2 and a mean of 0.2. In contrast, the correlation between each coverage metric and fault detection tends to be higher, with means and medians ranging from roughly 0.3 for *Blocking* coverage to roughly 0.7 for *Blocked* coverage. Additionally, several outliers, both above and below the mean, can be seen; for example in the near perfect correlation of *Blocked* coverage and fault detection for one mutant, and the very low (and sometimes even negative) correlations exhibited for a handful of combinations of coverage and mutant scenarios.

For each metric there exists at least one single fault object for which the correlation with fault detection is at or above 0.88. Further, even when coverage weakly correlates with fault detection, this

[3]For each boxplot, the mean is shown as a star, the box plot whiskers represent data within the 1.5 times the interquartile range, and the outliers are shown as red "+" marks. This convention is maintained for box plots shown in future sections.



(a) Arraylist

(b) Boundedbuffer

(c) Vector

Figure 3.4: Coverage versus fault detection effectiveness, mutation objects

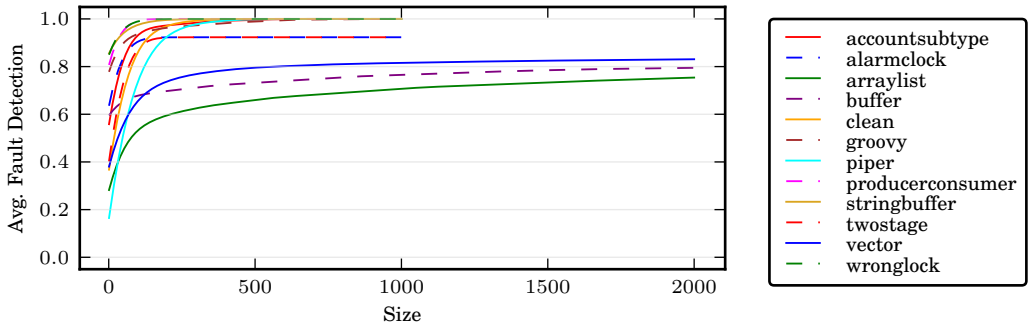


Figure 3.5: Size versus fault detection effectiveness, all objects

correlation is often higher than the correlation of fault detection and size (*S-FF*). These results provide evidence that each metric is a useful predictor of concurrency testing effectiveness, depending on program.

The best metric, however, varies across programs, and no single metric is a consistent predictor of effectiveness, though *PSet* is often quite strong. For the single fault programs, *PSet* shows the highest correlation for four programs among nine single fault programs in total, and *PSet* always shows high or moderate correlations except in the case of *Boundedbuffer*. Although *PSet* has a low average/median of 0.2 (*Boundedbuffer*), *PSet* has a better correlation than other coverage metrics.

The reason for this variation is unclear, but we believe this occurs because the metric's intuition does not always capture the single fault present. This is supported by the results shown in Figure 3.6, where we see a wide variation even within program depending on the mutant used. For example, for the *Vector* program, the relationship between coverage and fault detection varies strongly for several metrics, e.g., *Def-Use*, which varies from exhibiting a negligible relationship to a moderately strong relationship depending on the mutant used. This contrasts strongly with the very consistent relationships between coverage and size for most metrics when applied to all of *Vector*'s mutants.

Table 3.4: Correlations over coverage metrics
Each cell contains (coverage & fault detection effectiveness correlation, size & coverage correlation). S-FF denotes size & fault detection effectiveness correlation

	Blocked	**Blocked-Pair**	**Blocking**	**Def-Use**	**S-FF**
Accountsubtype	0.39, 0.11	0.39, 0.11	0.35, 0.10	0.60, 0.28	0.22
Alarmclock	0.77, 0.25	0.52, 0.24	0.27, 0.23	0.56, 0.22	0.05
Clean	0.16, 0.16	0.73, 0.23	0.19, 0.40	0.96, 0.29	0.30
Groovy	0.46, 0.36	0.50, 0.37	0.45, 0.37	0.45, 0.16	0.17
Piper	0.0, 0.0	0.62, 0.45	0.48, 0.25	0.07, 0.03	0.38
Producerconsumer	0.14, 0.03	0.17, 0.21	0.14, 0.16	0.57, 0.15	0.12
Stringbuffer	0.58, 0.18	0.67, 0.23	0.59, 0.31	0.43, 0.12	0.13
Twostage	0.88, 0.23	0.94, 0.13	0.88, 0.23	0.92, 0.13	0.10
Wronglock	0.12, 0.01	0.12, 0.01	0.12, 0.01	0.53, 0.13	0.11
	Follows	**LR-Def**	**PSet**	**Sync-Pair**	**S-FF**
Accountsubtype	0.28, 0.09	0.30, 0.12	0.57, 0.42	0.28, 0.09	0.22
Alarmclock	0.66, 0.29	0.59, 0.30	0.59, 0.35	0.19, 0.26	0.05
Clean	0.17, 0.42	0.91, 0.30	0.83, 0.28	0.09, 0.05	0.30
Groovy	0.52, 0.24	0.30, 0.09	0.48, 0.18	0.52, 0.24	0.17
Piper	0.59, 0.49	0.66, 0.27	0.67, 0.27	0.62, 0.45	0.38
Producerconsumer	0.21, 0.43	0.46, 0.26	0.30, 0.26	0.11, 0.20	0.12
Stringbuffer	0.44, 0.35	0.74, 0.14	0.87, 0.15	0.66, 0.23	0.13
Twostage	0.88, 0.23	0.95, 0.13	0.96, 0.13	0.96, 0.13	0.10
Wronglock	0.0, 0.0	0.50, 0.15	0.58, 0.21	0.0, 0.0	0.11

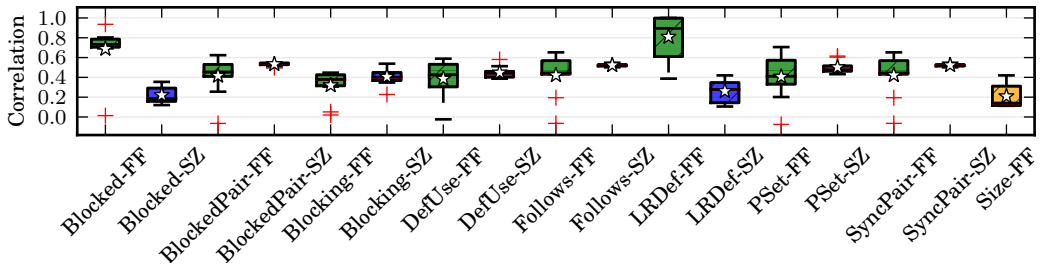
In any case, the variation in the best metric for a given object indicates that selecting an effective metric may be challenging. Additionally, the occasional low and often moderate correlation between coverage and fault detection (and somewhat surprisingly, size and fault detection) hints that factors other than those captured by the concurrency coverage metrics may relate to fault detection effectiveness. We discuss this further in Section 3.4.2.

3.3.3 Models of effectiveness

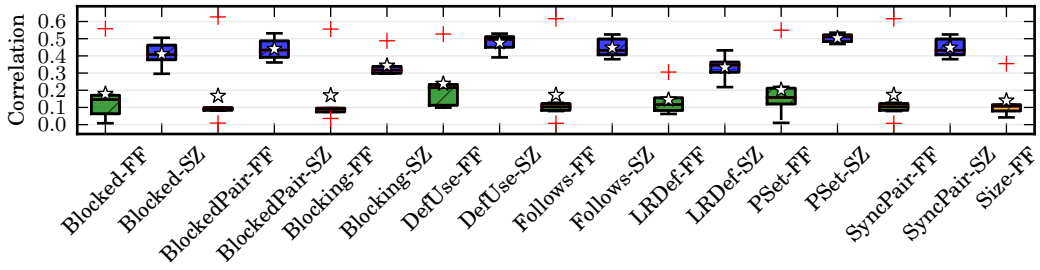
Based on the previous two analyses we can see that for every metric, coverage levels do correspond (somewhat) to testing effectiveness. However, we also see that test suite size and coverage are often similarly correlated, and thus the relationship between size, coverage and fault detection is unclear. It is possible that, in fact, coverage and size are not very independent of each other in terms of their effect on fault detection; for example, depending on the case example, either coverage or size alone may be a sufficient exploratory variable for fault detection.

Does coverage predict fault detection effectiveness, or merely reflect test suite size? And to what extent (if any) does considering coverage improvement increase the ability to predict fault detection? To address these questions we used linear regression to attempt to model how test suite size and coverage jointly influence the effectiveness of the testing process, with the goal of determining whether coverage has an independent explanatory ability with respect to fault detection.

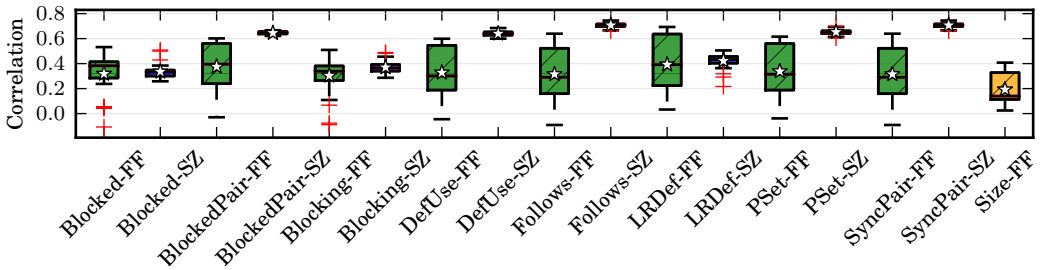
In linear regression, we model the data as a linear equation $y = \beta_1 x_1 + \beta_2 x_2 + \ldots + \beta_p x_p + \varepsilon_i$ where variables x_i correspond to explanatory factors and variable y denotes the dependent variable. After modelling the data, the coefficient of determination R^2 is produced. R^2 indicates how well the data fits the model, and can be interpreted as the proportion of variability explained by the model, e.g. a fit of 0.6 indicates about 60% of the variation can be explained by the explanatory variables. In many cases,



(a) Arraylist



(b) Boundedbuffer



(c) Vector

Figure 3.6: Correlations across mutants, mutation objects.
FF = fault detection, SZ = test suite size.

the goal of linear regression is *model selection*: from a set of candidate models, select the model that offers the highest *goodness of fit*, while omitting unneeded explanatory variables.

In our work, we will focus largely upon the *adjusted* R^2. Adjusted R^2 is a measure of fitness that adjusts for the number of explanatory variables. When comparing two models, a model with more explanatory variables will have a higher adjusted R^2 only when additional variables significantly contribute.[4] Strictly speaking, adjusted R^2 cannot be used to indicate the proportion of variance captured, but as adjusted R^2 is always less than or equal to R^2, we can infer that the proportion of variance captured by a model is equal to or greater than that given by adjusted R^2. Thus if for some model an adjusted R^2 of 0.6 is produced, this indicates that the model explains at least 60% of the variation in fault detection.

In this case we would like to model fault detection effectiveness for each object and coverage metric using test suite size (SZ) and/or coverage level (CV) as explanatory variables. If the best models always employ coverage levels as an explanatory factor, this indicates that coverage has an independent ability to predict fault detection effectiveness. Accordingly, for every combination of object and coverage metric where coverage varies, we fit all possible linear models employing combinations of SZ, $log(SZ)$, CV, and

[4] We also used Mallow's C_p to determine goodness of fit [62]. The results when using Mallow's led to the same conclusions, and we have presented results using adjusted R^2 as we believe this metric is easier to interpret.

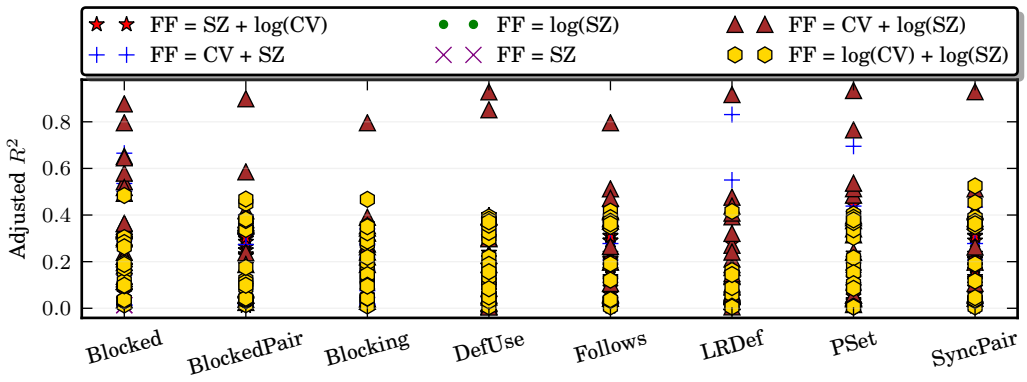


Figure 3.7: Adjusted R^2 for every best fit model, all combinations of objects & coverage metrics. FF = fault detection, SZ = test suite size, CV = % coverage.

log(CV) as explanatory variables (with fault detection (*FF*) as the dependent variable). Note that the use of *log* does not necessarily indicate that a factor is less important (in terms of fit) than a factor linearly related, but indicates that the relationship is logarithmic.

Our fitting process results in over 10,000 regression models and thus listing regression models with computed coefficients is infeasible; additionally, we are interested in exploring how well size and coverage levels model fault detection effectiveness, not the specific models. To summarize our data, we began by selecting the best fitting model for each object/coverage metric pair. We plot the associated adjusted R^2 in Figure 3.7 for each coverage metric, across all objects, indicating which set of explanatory variables had the highest fit. For example, we see that for the *Def-Use* metric, for two objects adjusted R^2 was greater than 0.8, indicating high fit with model $FF = \alpha \times CV + \beta \times log(SZ)$, while on all other objects fit was under 0.4, suggesting a low to moderate fit. Here we can clearly see the variation in metric effectiveness, with fits ranging from less than 0.2 to over 0.8, indicating a wide variation in predictive power. However, for all coverage metrics, for at least one object an adjusted R^2 of 0.8 or above was observed, indicating high fit, and for many objects fits above 0.4 were observed, indicating moderate fit.

Following this, we wished to measure the degree to which coverage improves the model fit, i.e., how much does adding coverage as a dependent variable improve the fit as compared to models using size alone? To answer this question, we computed minimum and maximum relative improvement in adjusted R^2 when using models with two dependent variables over models using size alone as a dependent variable. We list the results in Table 3.5 for single fault objects, and plot the results in Figure 3.8 for mutation objects. In the plots, the columns *MN* and *MX* represent the minimum and the maximum relative increase in adjusted R^2 when using two dependent variables for the corresponding object. An *NA* denotes that the improvement cannot be computed, as the linear regression's adjusted R^2 is 0.0 (resulting in infinite improvement).

As shown in Table 3.5, in many cases adjusted R^2 greatly improved with the addition of coverage to the regression models. In several instances, for example when applying nearly every coverage metric to the *Stringbuffer* object, we see improvements over 100%, indicating a more than double increase in adjusted R^2. In the case of mutation objects, we see less consistency, with *Arraylist* exhibiting small improvements (less than 10% increases), and *Vector* exhibiting a mix of small to moderates increases ranging from under 5% up to 30% (see Figure 3.8).

In some cases, however, the improvement found in using coverage as part of the regression model is small, indicating that test suite size is the main component of effective testing. For example, *Blocked*

Table 3.5: Minimum and maximum relative increase in adjusted R^2 when using two dependent variables.

	Blocked	**Blocked-Pair**	**Blocking**	**Def-Use**
Accountsubtype	0.0%, 45.8%	0.0%, 44.7%	0.0%, 34.0%	121.9%, 134.3%
Alarmclock	3293.5%, 3858.0%	1591.1%, 1767.3%	351.2%, 483.0%	1847.4%, 2008.1%
Clean	0.0%, 0.4%	67.0%, 122.9%	0.0%, 0.5%	244.6%, 253.7%
Groovy	198.5%, 313.9%	241.8%, 355.4%	182.8%, 280.5%	131.5%, 209.3%
Piper	NA	16.5%, 30.1%	0.0%, 13.0%	NA
Producerconsumer	0.0%, 10.4%	0.0%, 6.9%	0.0%, 6.5%	NA
Stringbuffer	369.1%, 562.9%	518.9%, 542.0%	386.1%, 540.3%	NA
Twostage	1384.0%, 1497.6%	1624.1%, 1703.9%	1384.0%, 1497.6%	1511.5%, 1609.3%
Wronglock	0.0%, 14.3%	0.0%, 14.3%	0.0%, 14.3%	223.5%, 245.1%
	Follows	**LR-Def**	**PSet**	**Sync-Pair**
Accountsubtype	0.0%, 20.2%	NA	104.4%, 116.5%	0.0%, 20.2%
Alarmclock	2576.8%, 2791.3%	2170.2%, 2446.5%	2211.9%, 2621.7%	138.1%, 179.3%
Clean	0.0%, 1.0%	199.8%, 216.5%	142.0%, 164.7%	0.0%, 0.1%
Groovy	257.7%, 279.2%	27.0%, 85.7%	169.2%, 228.0%	257.7%, 279.2%
Piper	6.3%, 20.5%	NA	43.6%, 55.3%	16.5%, 31.1%
Producerconsumer	0.0%, 5.2%	NA	0.0%, 32.5%	0.0%, 1.6%
Stringbuffer	166.2%, 296.9%	624.7%, 653.9%	927.4%, 948.7%	514.3%, 619.2%
Twostage	1384.0%, 1497.6%	1688.0%, 1740.2%	1724.8%, 1774.8%	1627.3%, 1764.8%
Wronglock	NA	NA	289.3%, 294.2%	NA

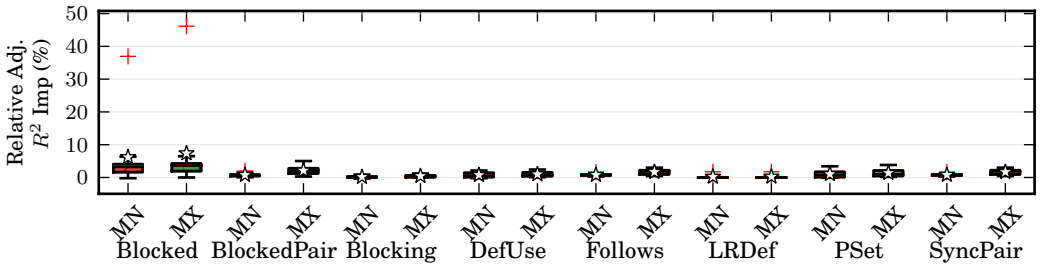
coverage applied to the *Clean* object yields a maximum improvement of only 0.4%, and for the *Bounded-buffer* object (Figure 3.8) we see several instances where the relative change in adjusted R^2 is negative, indicating that the addition of coverage to the model provides no statistically significant improvement to the predictive power of the model.

Based on these analyses, we can see that while no single set of explanatory variables is best, much of the time models based on both coverage and size are preferable to models using only one explanatory variable. Indeed, in several cases the addition of coverage to the model improves the model fit many times over. This provides evidence that coverage metrics have a predictive ability separate from test suite size. Nevertheless, the adjusted R^2 is generally less than 0.8, indicating that while our models do have reasonable predictive power, a significant proportion of variability is not accounted for by the models. Furthermore, in some cases coverage provides little or no predictive power, leaving test suite size as the sole (and often also weak, per Section 3.3.2) predictor of testing effectiveness. We discuss this further in Section 3.4.2.

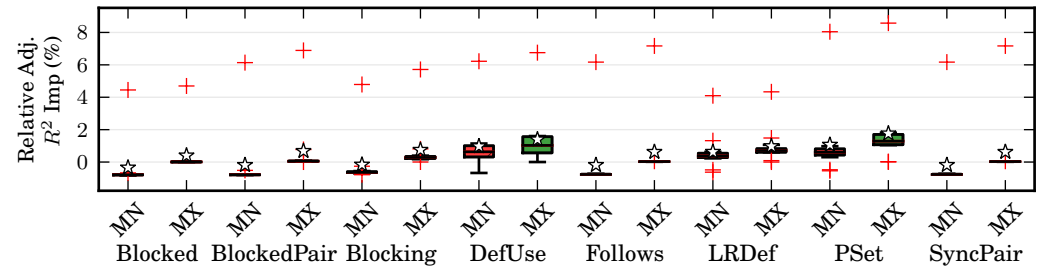
3.3.4 Effectiveness of maximum coverage

Our first three analyses have characterized the relationship between test suite size, coverage and fault detection effectiveness and statistically established that for each metric, coverage level has a predictive ability for fault detection apart from that of test suite size. From these results, we can see that while not every coverage metric is highly effective for all programs, all coverage metrics do appear to have value. Thus, it is worthwhile to use concurrency coverage metrics (in addition to test suite size) as methods for estimating the concurrency fault detection effectiveness of a testing process.

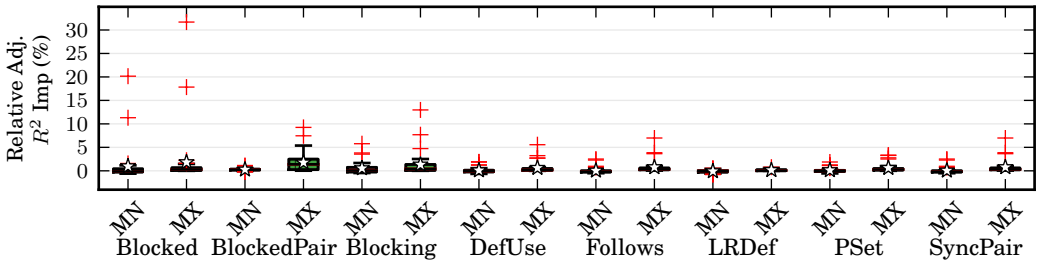
Per *RQ2*, however, we also would like to quantify the ability of test suites to quickly achieve high levels of concurrency coverage. To do this, for each program and coverage metric, we compared test suites of maximum achievable coverage, generated using a greedy algorithm described in Section 3.2.2,



(a) Arraylist



(b) Boundedbuffer



(c) Vector

Figure 3.8: Minimum and maximum relative increase in adjusted R^2 when using two dependent variables, mutation objects.

MN = minimum, MX = maximum.

against random test suites of equal size. Our expectation is that if a metric is a reasonable target for test case generation, holding the method of test case generation constant while reducing generated test executions to construct small, high coverage test suites should result in more effective test suites than pure random test case generation.

We began by formulating hypothesis H: test suites satisfying maximum achievable coverage will outperform random test suites of equal size in terms of fault detection. We evaluated H for each combination of program and coverage metric using a two-tailed bootstrapped paired permutation test, a non-parametric statistical test that calculates the probability p that two paired sets of data come from the same population [50]. The null hypothesis H_0 is that test suites achieving maximum achievable coverage are equally as effective as random test suites of equal size.

For each combination of coverage metric and object (per mutant for mutation objects), there are 100 test suites generated to achieve maximum achievable coverage (hereafter referred to as maximum coverage) (see Section 3.2.2). Each test suite was paired with a randomly selected test suite of equal size. Following this, the permutation test was applied using 250,000 permutations for each p-value [50]. Following the test, we computed the average fault detection when using test suites reduced to achieve

Table 3.6: Maximum achievable coverage test suite statistics
MFF = Maximum coverage fault detection, RFF = Random fault detection, Cv = % Increase in coverage over random, Sz = Test suite size (* = Not statistically significant difference at $\alpha = 0.05$)

	Blocked				Blocked-Pair				Blocking			
	MFF	**RFF**	**Cv**	**Sz**	**MFF**	**RFF**	**Cv**	**Sz**	**MFF**	**RFF**	**Cv**	**Sz**
Accountsubtype	0.19	0.06	31.9%	2.06	0.14	0.04	35.0%	2.16	0.09*	0.05*	29.5%	2.00
Alarmclock	0.92	0.34	54.0%	1.99	0.92	0.32	13.3%	2.20	0.29*	0.20*	81.4%	2.06
Clean	0.0	0.07	34.7%	1.93	0.0	0.10	0.0%	2.71	0.0	0.08	46.9%	2.3
Groovy	0.67*	0.64*	151.0%	3.72	0.63*	0.59*	182.4%	3.86	0.63	0.51	206.5%	3.4
Piper	0.00*	0.02*	0.0%*	1.0	0.39	0.03	13.9%	2.07	0.25	0.02	30.0%	1.96
Producerconsumer	0.21*	0.23*	5.4%	1.17	0.63	0.50	0.0%*	4.31	0.52	0.29	38.0%	2.13
Stringbuffer	0.78	0.53	168.4%	2.36	1.0	0.87	6.1%	6.50	0.97	0.62	209.5%	3.06
Twostage	0.92	0.16	431.9%	3.14	0.92	0.1	15.3%	3.2	0.92	0.1	405.0%	3.1
Wronglock	0.24*	0.26*	7.4%	1.0	0.21	0.35	3.1%*	1.0	0.26*	0.33*	2.3%*	1.0

	Def-Use				Follows				LR-Def			
	MFF	**RFF**	**Cv**	**Sz**	**MFF**	**RFF**	**Cv**	**Sz**	**MFF**	**RFF**	**Cv**	**Sz**
Accountsubtype	0.13	0.3	22.0%	2.99	0.24	0.06	7.1%	1.92	0.23	0.03	1.9%	1.87
Alarmclock	0.92	0.30	23.4%	3.51	0.52	0.26	62.3%	2.03	0.2*	0.27*	49.6%	2.01
Clean	1.0	0.04	5.2%	2.0	0.03*	0.08*	111.7%	1.28	0.03*	0.07*	14.3%	1.03
Groovy	0.35*	0.43*	5.3%	3.0	0.26	0.45	59.1%	3.02	0.30*	0.38*	6.3%	2.09
Piper	0.0*	0.02*	0.5%	1.13	0.70	0.09	13.0%	3.54	0.01*	0.03*	2.8%	1.78
Producerconsumer	1.0	0.36	4.1%	2.0	0.5*	0.5*	24.7%	3.71	1.0	0.31	5.9%	2.30
Stringbuffer	0.33	0.56	6.2%	2.33	1.0	0.83	238.1%	4.46	0.4*	0.30*	14.3%	1.4
Twostage	0.92	0.13	8.3%	2.92	0.92	0.07	374.5%	2.92	0.03*	0.03*	72.3%	1.19
Wronglock	0.34*	0.46*	19.5%	2.14	0.34*	0.35*	0.0%*	1.0	0.28*	0.33*	5.9%	2.0

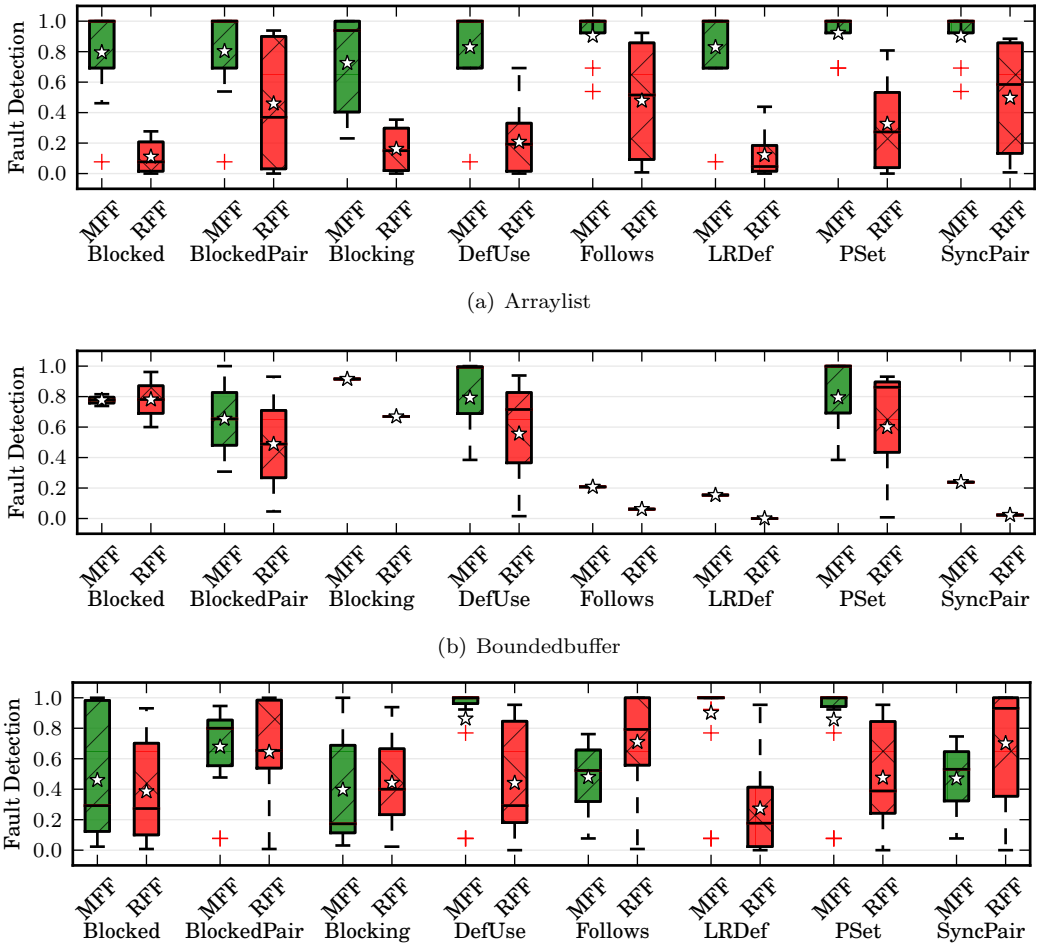
	PSet				Sync-Pair			
	MFF	**RFF**	**Cv**	**Sz**	**MFF**	**RFF**	**Cv**	**Sz**
Accountsubtype	0.36*	0.44*	29.4%	6.6	0.21	0.0	8.1%	1.87
Alarmclock	0.92	0.4	35.0%	5.20	0.53	0.26	14.9%	2.04
Clean	1.0	0.11	11.4%	2.93	0.06*	0.06*	8.7%	1.30
Groovy	0.33*	0.4*	6.8%	3.0	0.41*	0.46*	52.0%	3.02
Piper	0.43	0.06	5.1%	1.94	0.64	0.03	53.6%	3.49
Producerconsumer	1.0	0.4	6.3%	2.34	0.5*	0.38*	30.4%	3.72
Stringbuffer	1.0	0.76	7.3%	3.0	1.0	0.74	38.7%	4.35
Twostage	0.92	0.06	26.8%	2.92	0.92	0.07	66.6%	2.92
Wronglock	0.46	0.60	47.3%	2.96	0.31*	0.30*	0.0%*	1.0

maximum coverage, the average relative improvement in coverage over random test suites, and the average fault detection for the random test suites.

Table 3.6 lists the results of this analysis for objects with only a single fault. (Note that fault detection is the ratio of test suites detecting the fault to the total number of test suites.) Figure 3.9 plots the fault detection for greedily reduced test suites and random test suites of equal size across mutants as a boxplot. The column *MFF* represents the fault detection for the reduced test suites for each object and coverage metric studied, and the column *RFF* represents fault detection for random test suites of equal size. Figure 3.10 plots the relative increase in coverage when using greedy reduced tests suites over randomly generated test suites of equal size.

Our analysis results imply that achieving high coverage generally yields not only statistically significant, but also practically significant increases in fault detection: large, often twofold or more increases can be observed. For example, for the *Clean* object with the *Def-Use* coverage metric, the average fault detection of test suites achieving maximum coverage is generally higher (up to 25 times higher) than that of randomly generated test suites.

We can see a similar tendency for mutation object *Arraylist*. For the *Arraylist* object, the mean fault detection of maximum achievable test suites of every coverage metric is higher than or equal to the highest fault detection of corresponding randomly generated test suites.



(a) Arraylist

(b) Boundedbuffer

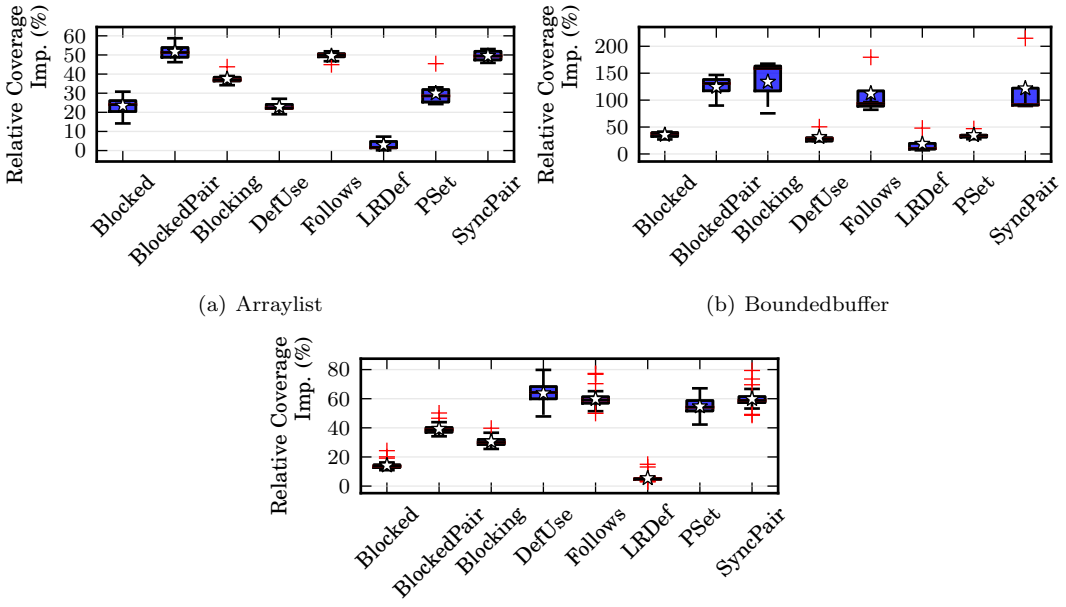
(c) Vector

Figure 3.9: Maximum fault detection, greedy versus random, across mutants. MFF = maximum fault detection, RFF = random fault detection.

Note that, for the *Boundedbuffer* object, the reduced test suites with respect to a coverage metric provide useful results although their correlations with fault detection are low. In contrast, *LR-Def* displays moderate to high correlations in fault detection as shown in Table 3.4, but the reduced test suites with respect to *LR-Def* do not have higher fault detection than randomly generated test suites in most cases.

We were surprised, however, that there were object/coverage metric pairs for which reduction to maximize coverage had a *negative* impact on the fault detection effectiveness of the testing process. For example, for *Wronglock*, test suites reduced to satisfy *Blocked-Pair* found the fault 21% of the time, as compared to 35% when using random test suites of equal size.

The case in which *Def-Use* was applied to *Stringbuffer* was more surprising. Here we see greedily reduced test suites detecting the fault 33% of the time on average, relative to the 56% detection rate for randomly reduced test suites of equal size. As we demonstrate in Section 3.3.6, however, when achieving maximum coverage for complex coverage metrics, there exist several difficult-to-cover test requirements that are satisfied only by specific test executions that do not necessarily detect a fault (see Table 3.10). During greedy test suite reduction, these executions must be selected to achieve maximum coverage, and are thus useless with respect to fault detection, but always present. We hypothesize that this is the cause of this unusual behavior.



(a) Arraylist

(b) Boundedbuffer

(c) Vector

Figure 3.10: Relative improvement in coverage, greedy versus random, across mutants

3.3.5 Effect of combining concurrency coverage metrics

In the previous subsections, we demonstrated that while every coverage metric has a meaningful value as a predictor of fault detection effectiveness and also as a target for test generation, there is strong variation in the relative usefulness of the coverage metrics for both purposes across target programs. This implies that identifying a single proposed concurrency coverage metric to use for testing an arbitrary target program may be unrealistic.

One possible solution for addressing this variability is to combine complimentary concurrency coverage metrics, mitigating the shortfalls of each [43, 109]. To determine whether this solution is effective, we created and studied the effectiveness of six combined coverage metrics. The rationale for selecting these metrics was detailed in Section 3.2.1, but in short these combinations were viewed as most likely to yield improvements over the original metrics.

Combined Coverage Metrics as Predictors.

We begin by examining the effectiveness of our combined metrics as predictors of testing effectiveness. In Table 3.7 and Figure 3.11, we present the correlation of coverage and fault detection effectiveness of the combined coverage metrics as compared to the original metrics they are derived from. Based on these results, we see that the combined metrics are a mixed bag in terms of improvements. Across the single fault objects, in 26 of the 54 combinations of combined metrics and objects, the combined metric achieves a correlation equal to or higher than the highest correlation observed from its composite original metrics. Typically in these cases the gains over the highest correlation observed from an original metric is small, but in some cases the gains over the lowest performing metric are quite high. For example, in the case of the *Arraylist* object, the lowest correlation in the combined coverage metric is upgraded from the original coverage metrics, whereas the highest correlation still remains. In the case of the *Wronglock* object, only data access metrics are effective predictors of fault detection, with all pairwise synchronization based metrics achieving no higher than an 0.12 correlation. Similar behavior also occurs for the *Accountsubtype*

Table 3.7: Correlations over combined metrics.
Each cell contains coverage & fault detection correlation. CM = combined metric correlation.

	Blocked-Pair+Def-Use			Blocked-Pair+PSet			Follows+Def-Use		
	CM	Blocked-Pair	Def-Use	CM	Blocked-Pair	PSet	CM	Follows	Def-Use
Accountsubtype	0.61	0.39	0.60	0.59	0.39	0.57	0.60	0.28	0.60
Alarmclock	0.60	0.52	0.56	0.65	0.52	0.59	0.52	0.66	0.56
Clean	0.38	0.73	0.96	0.21	0.73	0.83	0.73	0.17	0.96
Groovy	0.56	0.50	0.45	0.55	0.50	0.48	0.51	0.52	0.45
Piper	0.59	0.62	0.07	0.48	0.62	0.67	0.62	0.59	0.07
Producerconsumer	0.31	0.17	0.57	0.15	0.17	0.30	0.17	0.21	0.57
Stringbuffer	0.46	0.67	0.43	0.61	0.67	0.87	0.67	0.44	0.43
Twostage	0.92	0.94	0.92	0.88	0.94	0.96	0.94	0.88	0.92
Wronglock	0.53	0.12	0.53	0.58	0.12	0.58	0.53	0.0*	0.53
	Follows+PSet			Sync-Pair+Def-Use			Sync-Pair+PSet		
	CM	Follows	PSet	CM	Sync-Pair	Def-Use	CM	Sync-Pair	PSet
Accountsubtype	0.58	0.28	0.57	0.60	0.28	0.60	0.58	0.28	0.57
Alarmclock	0.25	0.66	0.59	0.27	0.19	0.56	0.55	0.19	0.59
Clean	0.20	0.17	0.83	0.07	0.09	0.96	0.66	0.09	0.83
Groovy	0.52	0.52	0.48	0.51	0.52	0.45	0.52	0.52	0.48
Piper	0.63	0.59	0.67	0.61	0.62	0.07	0.67	0.62	0.67
Producerconsumer	0.11	0.21	0.30	0.26	0.11	0.57	0.14	0.11	0.30
Stringbuffer	0.66	0.44	0.87	0.66	0.66	0.43	0.74	0.66	0.87
Twostage	0.92	0.88	0.96	0.90	0.96	0.92	0.96	0.96	0.96
Wronglock	0.58	0.0*	0.58	0.53	0.0*	0.53	0.58	0.0*	0.58

object. In these scenarios, the failure of synchronization based metrics is masked by the inclusion of data access metrics (notably *PSet*, which per Section 3.3.2 we found to be the single most effective original metric overall). For the *Wronglock* and *Accountsubtype* objects, the all combined coverage metrics shows the moderate correlations ($0.53 \sim 0.58$ for *Wronglock*, and $0.58 \sim 0.61$ for *Accountsubtype*).

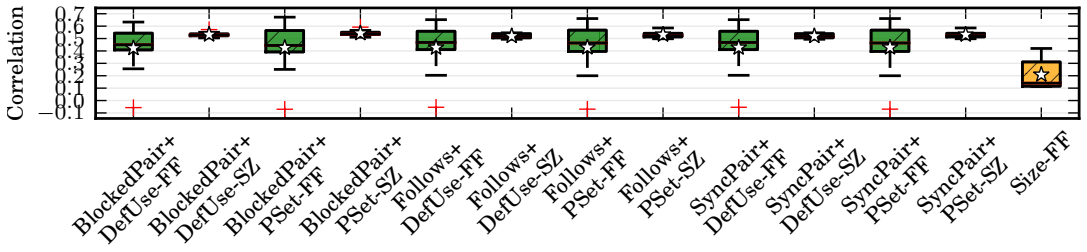
In the opposite scenario, however, where synchronization based metrics outperform data access metrics in terms of correlation, results are more mixed. For example, the combination of *Def-Use* to *Follows* results in a moderate correlation of 0.52, but this is a small drop from the original metrics' respective correlations of 0.56 and 0.66. In fact, examining our original suggestion of *PSet*, we find that for 23 of the 24 combinations of combined metrics including *PSet* and single fault objects, *PSet*'s correlation is within 0.05 of the combined correlation, and for 17 combinations it is equal to or greater than *PSet*'s correlation.

More concerning are scenarios where combinations of metrics significantly reduce the correlation. For example, in the case of *Follows+PSet*, the combined metric often performs far worse than either metric alone (e.g., *Alarmclock*, *Clean*, *Producerconsumer* all show the correlation dropping by 50%). Similar scenarios can be seen when using other combinations as well. Thus, while it is true that in some cases a combination of metrics can be a better predictor than single metrics alone, we cannot offer a general recommendation, as there are also many cases where combinations are less effective predictors.

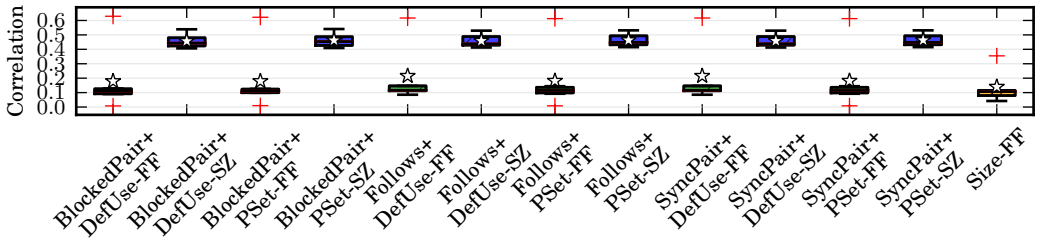
Combined Metrics as Test Case Generation Targets.

While having more effective predictors of testing effectiveness is useful, we are also interested in having more effective test case generation targets. In Table 3.8 and Figure 3.12 we present the fault detection results for test suites achieving the maximum achievable coverage for the single fault objects and for the mutation testing objects, respectively. In Table 3.9 we present the relative improvement in fault detection when using combined coverage metrics over the original coverage metrics for the single fault objects.

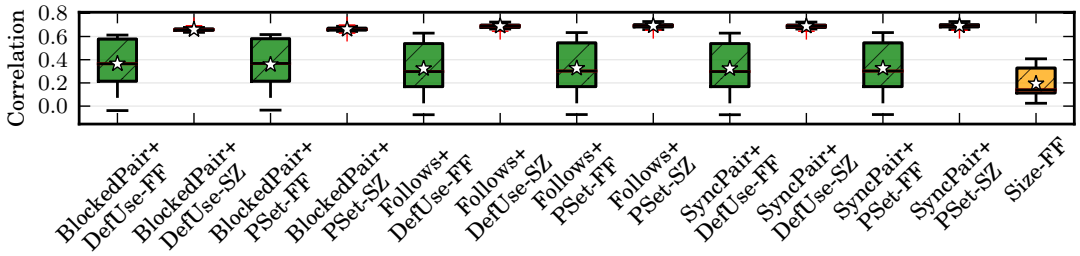
The results show that for every object and for every combined coverage metric, the fault detection



(a) Arraylist



(b) Boundedbuffer



(c) Vector

Figure 3.11: Correlations across mutants, combined metrics.
FF = fault detection, SZ = test suite size.

effectiveness of the reduced test suite with respect to a combined coverage metric is higher than or equal to that of an orignal coverage metric. Naturally, the fault detection for a given coverage metric can only remain the same or increase by combining it with another metric (the concurrency coverage metrics studied, like typical sequential coverage metrics, are monotonic). Therefore, the existence of improvements is not especially interesting.

Instead, we wish to determine whether combinations either offer improvements over both metrics simultaneously, indicating a clear improvement in fault detection for some objects and indicating less variability in the effectiveness of the metric as a test generation target; or alternatively, whether combinations offer improvements over each metric in different scenarios. In other words, we wish to determine whether, for some combined metric $A+B$, improvements are found over only A for one object, while improvements are found over only B for some other object.

Based on Table 3.9, we can see that statistically significant examples of both types of improvements exist. For example, when applying the *Blocked-Pair+PSet* coverage metric over the *Piper* object, improvements over *PSet* and *Blocked-Pair* of 62.5% and 78.4% exist.

Additionally, for the *Follows+Def-Use* combination, we can see that for both *Alarmclock* and *Clean*, the combined metric is an improvement over *Follows* by 76.4% and 3150.0%, while for the *Piper* and *Stringbuffer* objects it is a comparable improvement over *Def-Use*. Similar patterns can be seen for all other combinations of metrics, indicating that the combined metrics do frequently reduce variability as

Table 3.8: Maximum achievable coverage test suite statistics, combined metrics
MFF = maximum coverage fault detection, RFF = random fault detection, Cv = % increase in coverage over random, Sz = Test suite size (* = Not statistically significant difference at $\alpha = 0.05$)

	Blocked-Pair+Def-Use				Blocked-Pair+PSet				Follows+Def-Use			
	MFF	**RFF**	**Cv**	**Sz**	**MFF**	**RFF**	**Cv**	**Sz**	**MFF**	**RFF**	**Cv**	**Sz**
Accountsubtype	0.15	0.40	18.8%	3.28	0.36*	0.46*	23.7%	6.85	0.17*	0.28*	13.6%	3.12
Alarmclock	0.92	0.30	22.4%	3.88	0.92	0.45	32.0%	5.56	0.92	0.35	19.7%	4.32
Clean	1.0	0.18	11.6%	3.72	1.0	0.23	26.8%	4.16	1.0	0.11	4.1%	2.22
Groovy	0.65*	0.60*	17.3%	3.99	0.69*	0.58*	23.8%	4.00	0.30	0.41	10.0%	3.00
Piper	0.4	0.01	4.3%	2.06	0.7	0.02	18.7%	2.10	0.68	0.06	12.7%	3.59
Producerconsumer	1.0	0.60	8.2%	4.61	1.0	0.69	20.7%	4.83	1.0	0.55	7.4%	4.01
Stringbuffer	1.0	0.87	26.7%	6.89	1.0	0.89	69.0%	6.9	1.0	0.79	12.2%	4.38
Twostage	0.92	0.13	22.5%	3.76	0.92	0.16	228.0%	3.73	0.92	0.11	15.1%	2.92
Wronglock	0.34*	0.43*	17.7%	2.17	0.54*	0.56*	40.6%	2.97	0.41*	0.42*	17.1%	2.16
	Follows+PSet				**Sync-Pair+Def-Use**				**Sync-Pair+PSet**			
	MFF	**RFF**	**Cv**	**Sz**	**MFF**	**RFF**	**Cv**	**Sz**	**MFF**	**RFF**	**Cv**	**Sz**
Accountsubtype	0.36*	0.47*	19.3%	6.64	0.21	0.35	13.4%	3.14	0.4*	0.42*	18.5%	6.68
Alarmclock	0.92	0.46	13.6%	5.93	0.92	0.41	0.9%*	4.38	0.92	0.53	27.1%	6.00
Clean	1.0	0.08	9.5%	2.97	1.0	0.11	0.9%	2.17	1.0	0.07	7.6%	2.93
Groovy	0.46*	0.42*	15.7%	3.0	0.38*	0.46*	10.1%	3.01	0.33*	0.39*	14.6%	3.02
Piper	0.68	0.03	48.3%	3.57	0.70	0.08	18.9%	3.56	0.68	0.1	16.0%	3.53
Producerconsumer	1.0	0.55	52.5%	4.13	1.0	0.43	15.8%	3.99	1.0	0.55	10.5%	4.22
Stringbuffer	1.0	0.80	54.8%	4.56	1.0	0.72	12.8%	4.52	1.0	0.82	15.8%	4.61
Twostage	0.92	0.13	111.2%	2.92	0.92	0.09	0.0%	2.92	0.92	0.10	29.9%	2.92
Wronglock	0.52*	0.61*	41.6%	2.97	0.32	0.46	17.6%	2.14	0.48	0.6	43.0%	3.01

compared to the use of individual metrics.

This reduction in variability is further illustrated by examining the fault detection rates for original test suites (Section 3.3.4). While the fault detection effectiveness across combined metrics are consistent within each object, the fault detection effectiveness for original metrics sometimes vary strongly across metrics. For example, within pairwise metrics (i.e., those used to create combined metrics) test suites generated for the *Clean* object vary in average fault detection from 0.0 to 1.0 as shown in Table 3.6, while the average fault detection for combined metrics is always 1.0. Other objects exhibit similar behavior.

As noted in Section 3.3.4, there is no best original metric to use as a test case generation target. However, several combined metrics when used as test case generation targets always produce, on average, higher fault detection than any single original metric (excluding fault detection values which are not statistically significant). In fact, every combined metric containing *PSet* exhibits this behavior. Note that these test suites are typically larger than those generated solely from original metrics, but given the small size of all test suites (less than seven tests on average), this seems acceptable.

This result also supports our conjecture that there are other factors that influence testing effectiveness beyond those that the concurrency coverage metrics studied capture (see Section 3.4.2).

In summation, while the predictive value of combined metrics differs from that of original metrics in ways that is not necessarily positive or negative, combined metrics as test case generation targets — in particular those metrics based on a combination of *PSet* with a pairwise, synchronization metric — are clearly superior to any original metric studied.

3.3.6 Effectiveness of difficult-to-cover test requirements

Our analysis has clearly demonstrated that increasing coverage levels of the presented concurrency coverage metrics tends to result in practically significant increases in fault detection effectiveness. Nevertheless, this does not necessarily imply that all test requirements are worth the effort required to cover

Table 3.9: Relative improvement in fault detection using combined metrics
(* = Not statistically significant difference at $\alpha = 0.05$)

	Blocked-Pair+Def-Use		Blocked-Pair+PSet		Follows+Def-Use	
	Blocked-Pair	**Def-Use**	**Blocked-Pair**	**PSet**	**Follows**	**Def-Use**
Accountsubtype	5.2%*	11.1%*	147.3%	0.0%*	0.0%*	27.7%*
Alarmclock	0.0%*	0.0%*	0.0%*	0.0%*	76.4%	0.0%*
Clean	inf%	0.0%*	inf%	0.0%*	3150.0%	0.0%*
Groovy	2.4%*	84.7%	8.4%*	104.5%	14.2%*	0.0%*
Piper	1.9%*	inf%	78.4%	62.5%	0.0%*	inf%
Producerconsumer	56.6%	0.0%*	56.6%	0.0%*	100.0%	0.0%*
Stringbuffer	0.0%*	195.4%	0.0%*	0.0%*	0.0%*	195.4%
Twostage	0.0%*	0.0%*	0.0%*	0.0%*	0.0%*	0.0%*
Wronglock	60.7%	0.0%*	153.5%	18.3%*	20.0%*	20.0%*

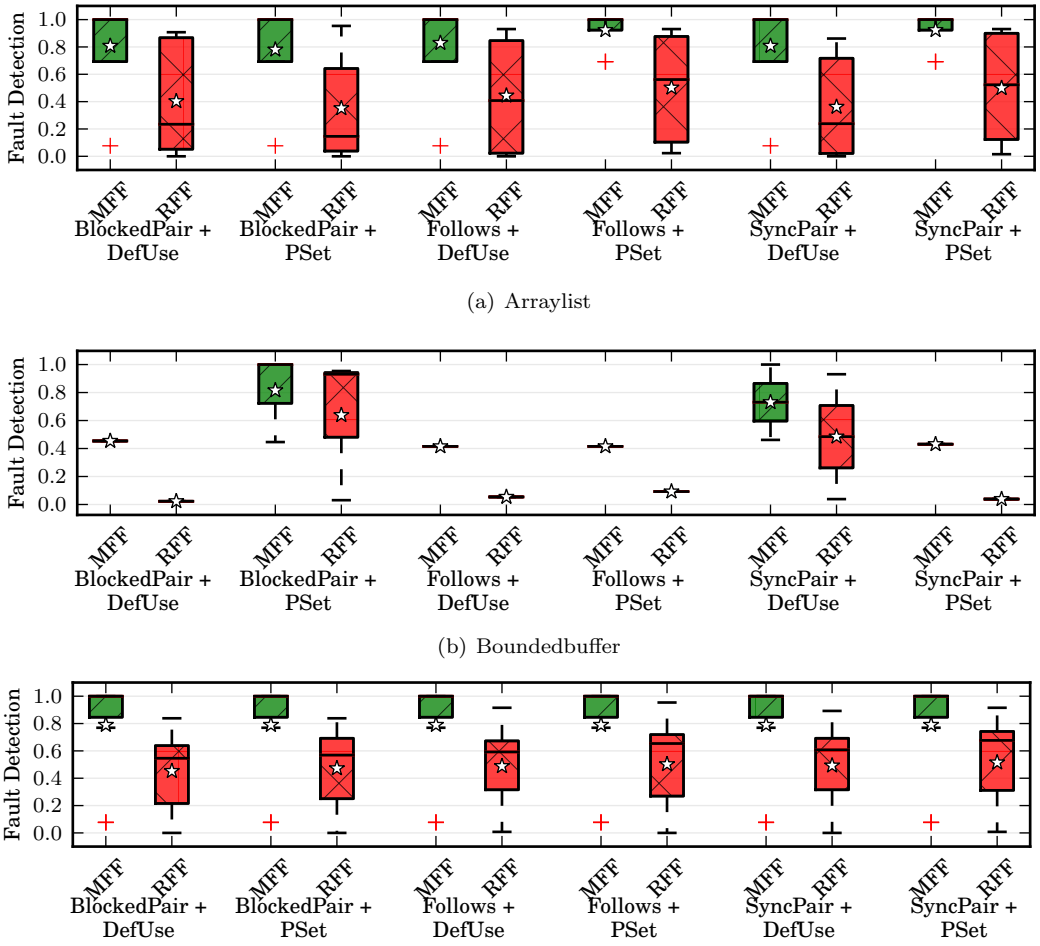
	Follows+PSet		Sync-Pair+Def-Use		Sync-Pair+PSet	
	Follows	**PSet**	**Sync-Pair**	**Def-Use**	**Sync-Pair**	**PSet**
Accountsubtype	50.0%	0.0%*	0.0%*	55.5%*	85.7%	8.3%*
Alarmclock	76.4%	0.0%*	71.4%	0.0%*	71.4%	0.0%*
Clean	3150.0%	0.0%*	1344.4%	0.0%*	1344.4%	0.0%*
Groovy	71.4%	36.3%*	0.0%*	8.6%*	0.0%*	0.0%*
Piper	0.0%*	58.9%	9.5%*	inf%	5.9%*	58.9%
Producerconsumer	100.0%	0.0%*	100.0%	0.0%*	100.0%	0.0%*
Stringbuffer	0.0%*	0.0%*	0.0%*	195.4%	0.0%*	0.0%*
Twostage	0.0%*	0.0%*	0.0%*	0.0%*	0.0%*	0.0%*
Wronglock	51.1%	13.3%*	2.4%*	0.0%*	53.6%	5.0%*

them. Per *RQ4*, we would like to determine whether difficult-to-cover test requirements — those that are satisfied by only a small percentage of tests — yield fault detection gains beyond those found in the other, easier to cover test requirements. This is key to establishing if specialized techniques which target hard to cover test requirements are likely to yield improvements in fault detection (akin to techniques for covering branches in structural coverage metrics).

First, we begin by establishing that difficult-to-cover test requirements exist. In Figure 3.13, we plot, for each covered test requirement, the percentage of test executions covering the requirement, i.e., difficulty of covering the test requirement (Figure B.3 for all objects, in Appendix). Requirements have been ordered from least likely to be covered, to mostly likely to be covered. (The x-axis represents the difficulty percentile, i.e., at 40% the requirement plotted is easier than 40% of all requirements and more difficult than 60%.) For each object and coverage criteria, there exists significant variation in the difficulty of covering test requirements – most objects contain several requirements that are covered by few executions (less than 1%), with most test executions being relatively easily covered (with greater than 10% covering the test executions).

Having established that difficult-to-cover test requirements exist, we would like to determine whether these test requirements are, on average, particularly effective at detecting faults. Towards this, in Table 3.10 we present the average fault detection of test executions covering difficult-to-cover requirements (defined as the 10% most difficult requirements to cover) as compared to other test requirements. We selected the 10% threshold as it frequently resulted in one or fewer test executions being selected, while larger thresholds were too easy to cover to be considered "difficult". Note that "NA" indicates that the number of requirements was less than 10, i.e. there was no bottom 10%.

Here we see that in some instances there does appear to be a practically and statistically significant



(a) Arraylist

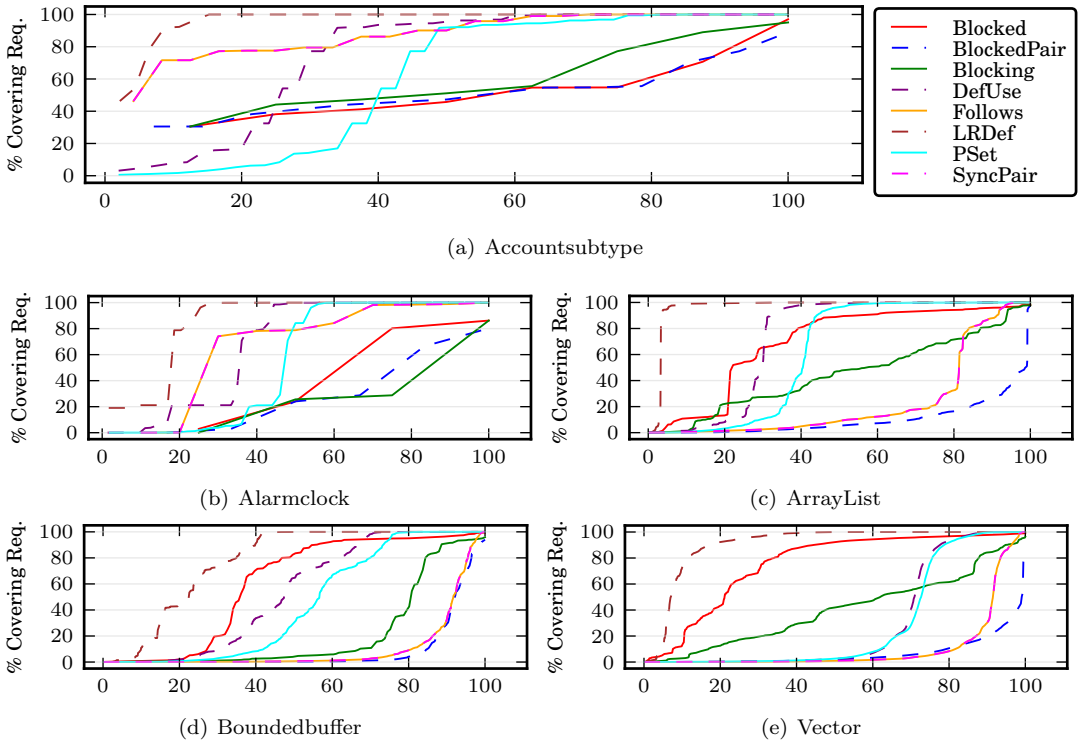
(b) Boundedbuffer

(c) Vector

Figure 3.12: Maximum fault detection, greedy versus random, across mutants, combined metrics. MFF = maximum fault detection, RFF = random fault detection.

difference in the fault detection rate of test executions satisfying difficult-to-cover test requirements relative to other test requirements. For example, for the *Arraylist* object, difficult-to-cover test requirements of all coverage metrics are better than other test requirements, with the relative differences in fault detection effectiveness ranging from 91.5% to 942.4%. Clearly, for many objects, the effort needed to satisfy difficult-to-cover requirements is potentially worthwhile.

In other cases, however, the relative difference between difficult-to-cover and easy-to-cover test requirements is either practically marginal (for example for the *Vector* where differences are small and often close to zero) or not statistically significant (for example the *Groovy* and *Stringbuffer* object). Given these results, it is difficult to draw any conclusions concerning the value of difficult-to-cover requirements in testing a particular program. In some cases the extra effort is clearly unlikely to be rewarded as the relative differences are minor. On the other hand, in many cases the relative difference is quite large, but (due to the small number of test requirements) not statistically significant. Thus it appears that studies with objects which produce larger numbers of test requirements are required to better address this question. We discuss the implications of this for concurrent test case generation approaches in the next section (see Section 3.4.4).



(a) Accountsubtype

(b) Alarmclock

(c) ArrayList

(d) Boundedbuffer

(e) Vector

Figure 3.13: Relative difficulty of covering individual coverage requirements for four single fault objects and all mutation objects.

The x-axis represents the difficulty percentile. Requirements covered by the fewest number of executions are leftmost on the x-axis; requirements covered by the largest number of executions are rightmost. The y-axis indicates the percentage of executions that cover the requirement.

3.4 Discussions

Our results have addressed our original research questions as follows. Per *RQ1* and *RQ2*, we have shown that for every coverage metric, for some programs (1) the metric is a moderate, independent predictor of fault detection, and (2) the testing process can be made more effective by using test suites that achieve maximum coverage instead of random test suites of equal size.

In short, we have provided evidence that existing concurrency coverage metrics can be useful. Consequently, testers can use concurrency coverage metrics as part of their testing process with confidence, either to estimate testing effectiveness, or as a goal for the testing process. Furthermore, testing researchers can justify as worthwhile the effort spent developing tools and techniques based on concurrency coverage metrics.

Nevertheless, the variation in the relative effectiveness of coverage metrics raises issues concerning how to apply these metrics in practice. Additionally, the generally moderate levels of correlation and fit observed hint that while these metrics appear effective, improvements to these metrics are both possible and desirable.

Towards addressing this variability and to better understand how test generation should be approached to improve fault detection, we proposed and addressed research questions *RQ3* and *RQ4*. Per *RQ3*, we have seen that using two coverage metrics combined can, in some cases, improve the reliability of coverage metrics as estimators of testing effectiveness and particularly as test generation targets. Per *RQ4* we have shown that at least in some cases, satisfying difficult-to-cover test requirements of-

Table 3.10: Fault detection effectiveness for difficult and easy to cover test requirements. DFF = difficult-to-cover fault detection, EFF = easy-to-cover fault detection, % = % increase in average fault detection for DFF over EFF coverage requirements (* = Not Statistically Significant at p = 0.05).

	Blocked			Blocked-Pair			Blocking		
	DFF	**EFF**	**%**	**DFF**	**EFF**	**%**	**DFF**	**EFF**	**%**
Arraylist	0.75	0.05	1259.3%	0.15	0.08	91.5%	0.46	0.04	942.4%
Boundedbuffer	0.19*	0.30*	0.0%*	0.23*	0.26*	0.0%*	0.17*	0.26*	0.0%*
Vector	0.13*	0.10*	31.8%*	0.06	0.08	0.0%	0.10*	0.09*	2.9%*
Accountsubtype	0.07*	0.07*	0.0%*	0.07*	0.07*	0.0%*	0.07*	0.07*	0.0%*
Alarmclock	NA	NA	NA	1.0*	0.28*	254.9%*	NA	NA	NA
Clean	NA	NA	NA	0.0*	0.0*	0.0%*	0.0*	0.0*	0.0%*
Groovy	0.34*	0.23*	46.5%*	0.34*	0.23*	47.6%*	0.34*	0.23*	50.2%*
Piper	NA	NA	NA	0.47*	0.04*	978.3%*	NA	NA	NA
Producerconsumer	NA	NA	NA	0.31	0.17	79.8%	0.21*	0.18*	18.8%*
Stringbuffer	NA	NA	NA	0.79*	0.50*	58.7%*	0.88*	0.49*	77.5%*
Twostage	1.0	0.10	854.7%	1.0*	0.21*	356.1%*	1.0	0.25	294.7%
Wronglock	NA	NA	NA	NA	NA	NA	NA	NA	NA

	Def-Use			Follows			LR-Def		
	DFF	**EFF**	**%**	**DFF**	**EFF**	**%**	**DFF**	**EFF**	**%**
Arraylist	0.10	0.05	106.1%	0.19	0.06	181.5%	0.18	0.04	345.5%
Boundedbuffer	0.39	0.30	30.0%	0.28*	0.25*	11.3%*	0.21	0.29	0.0%
Vector	0.04	0.07	0.0%	0.04	0.06	0.0%	0.11*	0.10*	19.5%*
Accountsubtype	0.07*	0.07*	5.3%*	0.07*	0.07*	0.0%*	0.07*	0.07*	0.0%*
Alarmclock	0.14*	0.16*	0.0%*	1.0*	0.17*	464.2%*	0.21	0.11	90.9%
Clean	0.5*	0.03*	1490.9%*	0.0*	0.00*	0.0%*	0.0*	0.03*	0.0%*
Groovy	0.23	0.21	8.1%	0.19*	0.22*	0.0%*	0.22*	0.21*	2.9%*
Piper	0.01	0.01	0.0%	0.19*	0.08*	123.5%*	0.01	0.02	0.0%
Producerconsumer	0.32*	0.18*	69.9%*	0.35	0.18	93.8%	0.67	0.19	248.3%
Stringbuffer	0.0	0.30	0.0%	0.0*	0.40*	0.0%*	0.16*	0.32*	0.0%*
Twostage	0.75	0.04	1611.9%	1.0*	0.19*	407.4%*	0.03	0.04	0.0%
Wronglock	0.28*	0.26*	4.7%*	NA	NA	NA	0.29*	0.26*	10.3%*

	PSet			Sync-Pair		
	DFF	**EFF**	**%**	**DFF**	**EFF**	**%**
Arraylist	0.16	0.06	176.5%	0.19	0.06	181.5%
Boundedbuffer	0.35*	0.32*	10.8%*	0.28*	0.25*	11.3%*
Vector	0.07	0.08	0.0%	0.04	0.06	0.0%
Accountsubtype	0.09	0.07	19.9%	0.07*	0.07*	0.0%*
Alarmclock	0.48*	0.26*	82.7%*	1.0*	0.17*	464.2%*
Clean	0.5*	0.02*	2289.7%*	0.0*	0.00*	0.0%*
Groovy	0.23*	0.21*	7.5%*	0.19*	0.22*	0.0%*
Piper	0.18	0.01	867.2%	0.19*	0.08*	123.5%*
Producerconsumer	0.39	0.19	107.2%	0.35	0.18	93.8%
Stringbuffer	0.5*	0.30*	65.5%*	0.0*	0.40*	0.0%*
Twostage	1.0	0.10	839.7%	1.0*	0.19*	407.4%*
Wronglock	0.31	0.27	13.0%	NA	NA	NA

ten returns meaningful improvements in fault detection. These results provide some guidance how test generation for concurrency testing can be improved with respect to the resulting fault detection rates.

In the remainder of this section, we discuss the practical implications of the study and highlight additional areas of research that we believe should be explored.

3.4.1 Practical implications for testers

Following a study of several coverage metrics, the question every tester naturally asks is: *which metric should* I *use?* Examining the correlation with fault detection (Table 3.4 and Figure 3.6) and the fault detection effectvieness of maximum test suite result (Table 3.6 and Figure 3.9), we see that if a tester must select a single "best" metric, *PSet* seems to be the only possible choice. For seven objects among nine single fault objects, *PSet* coverage's correlation with fault detection is over 0.57. *PSet* always achieved a greater correlation with fault detection than size (*S-FF*). Additionally, the reduced test suites with respect to *PSet* achieve higher fault detection than random test suites of equal size for six objects, and achieve lower fault detection than random test suite for only one object (*Wronglock*). *PSet* is clearly not ideal in many scenarios – *Def-Use* was similarly effective as a generation target for *Boundedbuffer* while requiring fewer test executions and *Blocking* was more effective as a generation target for *Groovy* – but on the whole it was consistently effective as both a predictor and for test case generation.

With respect to the other metrics, our results suggest basic guidelines. Recall from Table 3.2 the coverage metric properties of *singular/pairwise*. Comparing the results for *singular* and *pairwise* metrics while holding the other metric property (*synchronization/data access*) constant reveals two patterns.

First, the fault detection for maximum coverage test suites for pairwise metrics tends to be equal to or higher than when using singular metrics. Thus as test case generation target, it is preferable to select pairwise metrics. Second, pairwise metrics generally have higher correlation with fault detection and more reliable overall tendency across programs than singular metrics. For every single fault object, the correlation of *Blocked-Pair* is higher than or equal to the correlations of its singular versions *Blocked* and *Blocking*. In contrast, *LR-Def* often shows as high correlations as *Def-Use* or *PSet* do. But, the maximum test suites of *LR-Def* shows significantly less fault detection than *Def-Use* and *PSet*, which indicates its practical limitation.

Of course, as noted previously, pairwise metrics have more requirements, and thus require more test executions to achieve maximum coverage. Nevertheless, the stronger correlation between pairwise coverage metrics and fault detection indicates that investing the effort needed to satisfy a pairwise coverage metric is preferable to investing the same amount of effort satisfying a singular metric. When a test reaches a likely saturation point in a singular coverage metric, we recommend achieving as many pairwise coverage requirements as possible rather than targeting a few remaining singular requirements.

The above advice relates to the previously proposed individual metrics. Based on the results given in Section 3.3.5 related to *RQ3*, if we are primarily interested in selecting a test generation target, we would do well to use combined metrics. While the correlations for combined metrics, shown in Table 3.7, are not always improvements over those for the original metrics, fault detection rates for test suites achieving maximum coverage are typically improved. In particular, we recommend a metric combining *PSet* and a pairwise synchronization coverage metric (e.g., *Follows*), as this provides a somewhat reliable testing estimator and more effective test generation target than any of the original metrics used. As with the move from singular to pairwise metrics, this increases the number of requirements (being a combination of two pairwise metrics), but as shown in Table 3.8, for the systems studied the size of the resulting test suites is not significantly larger than the size of suites defined over the original metrics.

A final note: for some objects, there was a large difference in fault detection depending on the code constructs (*synchronization/data access*) used to define the metrics. For example, when using data access based coverage metrics with *Wronglock*, the correlation with fault detection was roughly four times that of synchronization based metrics. However, for *Piper* the opposite was true; data-access based metrics show poor fault detection in the reduced test suites. Even among combined metrics, which

are intended to reduce these variations by combining metrics based on different constructs, this behavior was still observed, for example, *Follows+PSet* as compared to *Blocked-Pair+PSet* for the *Arraylist* and *Boundedbuffer* systems.

We found this surprising: while in theory such behavior can also exist between foundationally different sequential coverage metrics (e.g., metrics defined over def-use pairs versus those defined over branch constructs), in our experience such dramatic differences do not occur in practice.

3.4.2 Limitations of existing concurrency metrics

As noted, in some cases the concurrency coverage metrics explored exhibited low correlation with fault detection and/or poor fit during linear regression. These results stand in sharp contrast to results related to sequential coverage criteria, where for example much better linear regression fit has been achieved using only test suite size and coverage levels, with adjusted R^2 values over 0.90 being typical [4, 69]. In contrast, we observed few adjusted R^2 values greater than 0.8, indicating that a great deal of effectiveness is unaccounted for by test suite size and coverage. By uncovering additional factors that contribute to fault detection effectiveness, we may be able to improve our concurrency coverage metrics and testing techniques.

As an initial step towards this, we extended our linear regression analysis to consider two additional factors: the probability of a delay being inserted (PB), and the length of the delay inserted (DL) (see Section 3.2.2). These factors were controlled for during test execution, and have been observed to impact the effectiveness of concurrent testing in previous work [41,51]. We then repeated our regression analysis, selecting the model with the highest fit for each combination of coverage metric and program.

Following this, we compared each selected model's fit against the same model with PB and DL omitted as explanatory variables. We found that while sometimes the improvement when using PB and DL as explanatory variables was small (< 0.01), often the improvement was significant: the average relative increase in adjusted R^2 was 50.5% (maximum 814%) and the average improvement in adjusted R^2 was 0.05 (maximum 0.37). In some cases, PB and DL account for the bulk of the predictive power; for example, for *Alarmclock* the best adjusted R^2 for the (usually effective) *PSet* metric increased from 0.45 to 0.78, an improvement of 75.1%.

We believe these results highlight the need to further improve concurrency coverage metrics to provide better guidance to testers and testing techniques. Ideally, a coverage metric should perfectly capture the effectiveness of the testing process, providing a highly accurate estimate of testing effectiveness, upon which techniques for improving coverage can be built. At a minimum, we would like concurrency coverage metrics to be better predictors than PB and DL, as the most effective set of parameters — much like the metrics explored — varies unpredictably depending on program.

3.4.3 Relation between metric effectiveness and fault type

One potential factor that may account for the variability in testing is the types of faults present. Concurrency faults, in contrast to sequential faults — which can take nearly any form — are errors in specific constructs: for example, data races, e.g., unsynchronized accesses to a shared variable with at least one write operation, and deadlocks, e.g., incorrect synchronization orders such as `wait(m)` after `notify(m)`. Thus, detecting these faults can be easier or more difficult depending on the metric used, as different metrics focus on different code constructs.

To investigate this, in Table 3.11 we again present the best metrics, as measured by correlation and

Table 3.11: Relation between fault types and concurrency coverage metrics

Fault type	Study object	Coverage metrics of highest correlation w/ fault detection	Coverage metrics of highest fault detection with maximum test suites
Atomicity violation	Stringbuffer	PSet (LR-Def)	Blocked-pair, Follows, PSet, Sync-pair
	Twostage	PSet, Sync-Pair, (Blocked, Blocked-Pair, Blocking, Def-Use, Follows, LR-Def, Sync-Pair)	Blocked, Blocked-Pair, Blocking, Def-Use, Follows, PSet, Sync-Pair
Data race	Accountsubtype	Def-Use	Follows
	Alarmclock	Blocked	Blocked, Blocked-pair, Def-Use, PSet
	Wronglock	PSet	NA
Deadlock (with wait)	Clean	Def-Use (Blocked-Pair, LR-Def, PSet)	Def-Use, PSet
	Groovy	Follows, Sync-Pair	Blocking
	Piper	PSet	Follows
Order violation	Producerconsumer	Def-Use	Def-Use, LR-Def, PSet

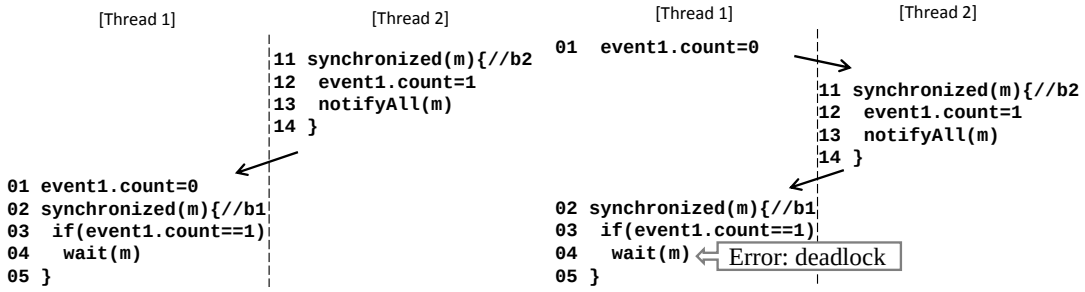
the effectiveness of maximum coverage test suites, for each object grouped by the type of fault present. The best metrics with respect to correlation are presented in the third column, while the best metrics with respect to fault detection rate for maximum achievable coverage test suites are presented in the fourth column ("NA" indicates no metric was better than random with statistical significance) [5]. In the case of ties for best, all metrics are presented. Furthermore, in the case of correlation, all metrics achieving high correlation (> 0.7) are listed in parentheses. Note that we present only the single fault objects as the type of faults present are already known from previous work [26,71,75]; when using mutation operators, we cannot be certain of the type of fault without a large amount of effort, an infeasible task for each mutant. Additionally, note that this (like the previous subsection) is an exploratory ad-hoc analysis; additional work will be required to verify the observations made.

Our expectation was that if the test requirements of a coverage metric M are formulated over constructs matching those involved with fault type T, metric M should perform well over objects of exhibiting fault type T. For example, we expected that *Def-Use* and *PSet* should perform well over objects exhibiting data race and atomicity violations, as the test requirements generated by these coverage metrics are based upon data access operations. We also expected that *Blocked-Pair*, *Follows*, and *Sync-Pair* metrics should perform well on objects exhibiting deadlock faults, as the test requirements of these coverage metrics are based on lock operations.

As shown in Table 3.11, there is no clear relationship between the fault type and the most effective coverage with respect to correlation. For example, for data race faults, *Def-Use*, *Blocked*, and *PSet* have the highest correlations on *Accountsubtype*, *Alarmclock*, *Wrongclock*, respectively. Indeed, even the best type of metric (synchronization/data access) varies depending on the program. Clearly, there is no best coverage metric for any fault type.

We see similar results with respect to fault detection effectiveness for maximum coverage test suites. For example, for deadlock faults, *Def-Use* and *PSet* have the highest fault detection with maximum test suites for *Clean*. However, for *Groovy* and *Piper*, *Blocking* and *Follows* have the highest fault detection with maximum achievable coverage test suites, respectively. Again, not only is there no best metric,

[5] To select the best metric with respect to fault detection, we exclude coverage metrics whose fault detection is not statistically significantly different than randomly generated test suites of equal size



(a) a correct execution that covers a *Sync-Pair* test requirement $\langle b2, b1 \rangle$

(b) an execution that covers a *Sync-Pair* test requirement $\langle b2, b1 \rangle$, which raises a deadlock

Figure 3.14: Two execution scenarios of *Clean*

there is no best type.

One possible reason why we observed no relationship between fault type and concurrency coverage metrics is because test requirements for concurrency coverage metrics do *not* capture concurrency faults *precisely*. To better understand why, consider Figure 3.14. In the figure, (a) and (b) show two executions that cover *Sync-Pair* requirement $\langle b2, b1 \rangle$ (i.e., a synchronization block $b2$ happens before a $b1$) where $b2$ is a synchronized block of Thread 2 (lines 11 to 14) containing `notifyAll(m)` and $b1$ is a synchronized block of Thread 1 (lines 2 to 5) containing `wait(m)`. Since `wait(m)` and `notifyAll(m)` should be used inside a synchronized block on m, we expect to detect the deadlock caused by calling `wait(m)` after `notify(m)` by covering the test requirements for *Sync-Pair* coverage, including $\langle b2, b1 \rangle$. However, no test requirement for *Sync-Pair* coverage is guaranteed to capture the deadlock situation precisely, as shown in Figure 3.14. In this case, both Figure 3.14(a) and (b) cover $\langle b2, b1 \rangle$, but only Figure 3.14(b) raises a deadlock.

In contrast, to detect this specific deadlock fault, the sequence of data accesses on the variable `event1.count` is more important than the sequence of lock operations. Figure 3.14 shows that the fault appears when Thread 1 executes a waiting operation on the lock `m` (line 04) after Thread 2 executes a notification on the same lock (line 13). The fault detection depends on the sequence of data accesses on `event1.count` (i.e., line 01 $\rightarrow$ line 12 $\rightarrow$ line 03). We suspect that this is the reason that the data access coverage metrics PSet and Def-Use show high correlation with the fault detection for *Clean*. This case implies that not only the coverage metric that captures a faulty thread interaction is important for fault detection, but also the coverage metric that captures execution paths up to the faulty thread interaction is important.

Such issues on concurrency coverage metrics again highlight the need to better understand how to capture what represents effective testing. Additionally, they help explain why using multiple concurrency coverage criteria, per Section 3.3.5, can be an effective strategy to improve fault detection.

3.4.4 Implications for concurrent test generation research

Work on test case generation methods for concurrency testing is an active — but relative to work on sequential testing — young area of research. In sequential test case generation, several techniques focus on methods for satisfying difficult-to-cover test requirements (e.g. symbolic execution, genetic approaches), and many, if not most approaches center around a single metric, branch coverage. In contrast, current approaches to concurrent test generation have little ability to target specific difficult-to-cover requirements, and the coverage metric used to evaluate these approaches has not been standardized.

Given this, it seems reasonable to consider whether, as in sequential testing, effort to develop new techniques for covering difficult-to-cover requirements is warranted, and if so what coverage metric(s) should be targeted. We have already largely addressed the latter question above in Section 3.4.1: *PSet*, combined with any of three pairwise, synchronized metrics already proposed, offers the most consistently high levels of fault detection. As noted previously in Section 3.3.3 and 3.4.3, however, there exist additional factors that current concurrency coverage metrics fail to capture. Thus, future work on concurrent test generation could be greatly improved by first considering how we can better (or perhaps more consistently) capture effective concurrent testing as a metric.

The answer to the former question — whether to target difficult test requirements — is similarly ambiguous. Given our results for *RQ4*, it seems that while in some cases difficult requirements do offer improved fault detection relative to other requirements (e.g. for the *Arraylist* object), in most cases no statistically significant improvements were found. Nevertheless, no statistically significant decreases in fault detection were observed, and thus if a test generation method could be found that increased the likelihood of satisfying difficult requirements, it would certainly improve testing effectiveness. Of course, the details of any new technique — specifically, whether the technique would slow the overall rate of test case generation — would determine whether it represents an improvement over existing approaches; there is little doubt that the potential to improve fault detection via targeting of difficult requirements exists.

3.5 Summary of this chapter

In this work, we have evaluated the relationship between eight previously proposed concurrency coverage metrics and fault detection effectiveness using twelve concurrent programs drawn from previous work in concurrency testing. We observed moderate correlations between coverage and fault detection effectiveness, established via linear regression that each coverage metric has a predictive value separate from test suite size, and found statistically and practically significant increases in fault detection effectiveness when using test suites reduced to achieve maximum coverage relative to random test suites of equal size. In addition, we confirmed that combinations of these coverage metric provide more reliable performance across different programs, particularly with respect to test generation, and that difficult-to-cover test requirements may be particularly effective with respect to fault detection. These results demonstrate that existing concurrency coverage metrics — in particular combinations of *PSet* and a pairwise synchronization based coverage metrics — can be effective metrics for evaluating concurrency testing effectiveness, and thus provide key evidence supporting the construction of techniques based on these metrics.

Nonetheless, while each metric explored was useful in some contexts, the predictive and test case generation value of each metric, even combined metrics which were proposed specifically to avoid this variation, often varied considerably from program to program, indicating that more work in this area is required. We hope to explore methods for improving these metrics in the future and encourage others to do the same.

Chapter 4. Test Generation Using Concurrency Coverage Metrics

4.1 Introduction [1]

Research on utilizing coverage criteria (e.g., branch coverage) to measure the quality of sequential program tests has been very active because of the strong correlation between test suites with high coverage and the fault-detection ability of those test suites [15, 69, 78]. For concurrent programs, concurrency coverage metrics (e.g., sync-pair coverage and statement-pair coverage [11, 102, 115]) have been proposed to capture the interleaving behaviors between multiple threads and the strong correlation between concurrency coverage and fault-detection ability is empirically shown [43]. However, there has been little research that aims to achieve high concurrency coverage to detect faults more effectively.

CUVE (Combinatorial concUrrent coVerage based tEst generation) is a technique that alleviates the limitations of the conventional testing techniques and detects fault effectively and efficiently. First, I have defined a new concurrency coverage criterion, *combinatorial concurrency coverage* (Section 4.2). Combinatorial concurrency coverage captures more diverse interleaving executions than conventional concurrency coverage. Second, I have developed a new testing technique *CUVE*, which generates diverse test executions fast by utilizing the combinatorial concurrency coverage (Section 4.3). I have performed a series of experiments to evaluate the effectiveness and efficiency of CUVE over conventional techniques by testing 65 mutated faulty versions of three Java programs and six real-fault Java programs (Section 4.4). In addition, I have compared CUVE with 12 random noise-injection techniques, a random scheduling technique, a systematic testing technique, and three bug pattern-directed testing techniques. The results demonstrate that CUVE is more effective and more efficient than any of the other testing techniques on most subjects (Section 4.5). Last, I analyze the experiment results in detail (Section 4.6).

4.2 Combinatorial concurrency coverage

4.2.1 Definition

A set of test requirements for the *combinatorial coverage* of a coverage C in a target program P is defined as follows:

$$\forall r_i, r_j \in TR(C) : r_i \neq r_j \leftrightarrow \{r_i, r_j\} \in TR(\mathbb{C}(C))$$

where $TR(C)$ is a set of test requirements of C for P and $\mathbb{C}(C)$ is a combinatorial coverage of C for P (from here on, we call C as a 'singular' coverage to distinguish from the combinatorial coverage and omit P if context is clear). In other words, a test requirement for the combinatorial coverage of C is defined as a set of two distinct test requirements for C. For example, suppose that a Sync-Pair coverage [41] for a target program P has four test requirements $\{r_1, r_2, r_3, r_4\}$. The test requirements of the corresponding combinatorial coverage are $\{\{r_1, r_2\}, \{r_1, r_3\}, \{r_1, r_4\}, \{r_2, r_3\}, \{r_2, r_4\}, \{r_3, r_4\}\}$. In general, for n test requirements of a singular coverage, we have $C(n, 2) = n \times (n-1)/2$ combinatorial test requirements.

[1] A part of this chapter was presented in ISSTA 2012 [41]

The main motivation for constructing the combinatorial coverage is that sets/combinations of two test requirements for a coverage criterion C can capture more diverse behaviors of P. For example, suppose that we have four test executions $\sigma_1, \sigma_2, \sigma_3$, and σ_4 for P which cover r_1, r_2, r_3, and r_4, respectively. Each of these test executions may not cover two different test requirements together (e.g., $\{r_1, r_2\}$ or $\{r_1, r_3\}$). [2] Thus, we have to generate more diverse test executions to cover the test requirements of a combinatorial coverage of a singular coverage C than to cover the test requirements of C. In order to generate more diverse test executions to cover the combinatorial test requirements for a singular coverage C, we have developed the *combinatorial concurrency coverage*. In particular, we utilize a combinatorial concurrency coverage (calling it CC) of the union of Sync-Pair metric (SP) [41] and Def-Use metric (DU)) [98]. In other words,

$$\forall r_i, r_j \in TR(SP) \cup TR(DU) : r_i \neq r_j \leftrightarrow \{r_i, r_j\} \in TR(CC)$$

SP has a test requirement for every pair of two synchronized blocks (i.e., two locking operators), which is satisfied when the two synchronized blocks consecutively holds a same lock. DU has a test requirement for every pair of write-read (and/or write-write [98]) operators that manipulate a same variable consecutively.

There are the two reasons to use the union of SP and DU to define the combinatorial concurrency coverage. First, a recent empirical study on concurrency coverage metrics shows that SP and DU show strong correlation with fault detection [43]. Second, targeting the test requirements of SP and DU together detects more faults than targeting the test requirements of SP and DU separately [43]. Note that a set of SP test requirements and a set of DU test requirements do not overlap with each other because test requirements of SP are defined over synchronization operators and test requirements of DU are defined over data access operators.

4.2.2 Advantages of the combinatorial concurrency coverage

In this section, we explain the advantages of the combinatorial concurrency coverage over singular concurrency coverages with examples. First, we show the shortcomings of a (singular) concurrency coverage in detecting concurrency errors. The previous empirical study on concurrency coverage metrics shows that tests achieving all feasible (singular) concurrency coverage requirements can still miss concurrency errors [43]. Then, we show how the combinatorial concurrency coverage metric can solve these issues.

Atomicity violation error

An atomicity violation error can occur when using the three statements that access the same shared variable by two different threads. Figure 4.1(c) shows an atomicity violation error, which may not be detected by covering test requirements of DU, but can be detected by covering test requirements of the combinatorial DU coverage.

The program has $Thread1$ that reads `x` (line 1) and then writes `x` (line 2) and $Thread2$ that writes `x` (line 3). We also present the last statement that writes `x` before the statements causing atomicity violation error (line 0). The set of the DU test requirements for the example is as follows:

$$\{< 0, 1 >, < 0, 2 >, < 0, 3 >, < 2, 3 >, < 3, 1 >, < 3, 2 >\}$$

[2] We say that an execution σ covers/satisfies a combinatorial requirement $\{r_1,r_2\}$ (denoted by $\sigma \models \{r_1,r_2\}$) if σ covers r_1 and r_2 (the order of covering r_1 and r_2 does not matter).

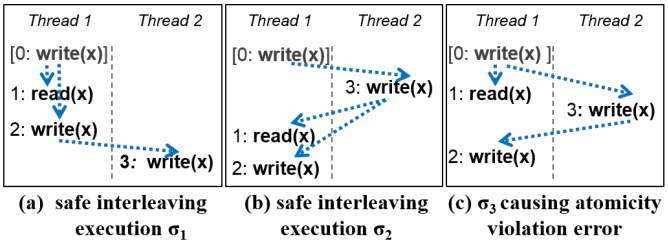


Figure 4.1: Example of atomicity violation error

```
arr[0..1] // array of size 2
len=2 ; // length of arr
p=0 ;

11 thread1() {   21 thread2() {     31 thread3() {
12   lock(m);    22   lock(m);       32   lock(m);
13   if(p+1<len) 23   if(p < len)    33   z = arr[p];
14     p++;      24     arr[p++]=y;  34   if(p > 0)
15   unlock(m);} 25   unlock(m);}    35     p--;
                                     36   unlock(m);}
```

Figure 4.2: Example of general race error

σ_1 (Fig. 4.1(a)) and σ_2 (Fig. 4.1(b)) cover all DU test requirements (i.e., σ_1 covers $< 0, 1 >$, $< 0, 2 >$ and $< 2, 3 >$ and σ_2 covers $< 0, 3 >$, $< 3, 1 >$ and $< 3, 2 >$) and do not raise an atomicity violation error. Thus, singular DU coverage based testing misses the error because the testing process will not generate σ_3 after it generates σ_1 and σ_2 which already covered all DU test requirements.

In contrast, a combinatorial DU requirement {¡0,1¿, ¡3,2¿} captures the erroneous execution because this combinatorial DU requirement cannot be covered by σ_1 nor σ_2, but by σ_3 that causes the atomicity violation error.

General race error

Figure 4.2 shows a concurrent program that has an out-of-bound array access fault (line 33). This error occurs when `thread1()` (lines 12-15), `thread2()` (lines 22-25), and `thread3()` (lines 32-36) are executed in order (i.e., when `p` is 2 at line 33). This fault cannot be detected by using bug pattern-directed techniques such as data race detector [84], atomicity violation detector [32], and atomic-set serializability violation detectors [39] because all data accesses are protected by the lock `m`, and also executions involve three threads instead of just two.

Moreover, test executions that cover all SP requirements of the program can still miss the error. For example, consider the set of the SP requirements for the program in the following:

$$\{< 12, 22 >, < 12, 32 >, < 22, 12 >, < 22, 32 >, < 32, 12 >, < 32, 22 >\}$$

Suppose that a singular SP based test process generated the following executions:

σ_0: lines 12–15; lines 32–36; lines 22–25

σ_1: lines 22–25; lines 32–36; lines 12–15

σ_2: lines 32–36; lines 12–15; lines 22–25

σ_3: lines 22–25; lines 12–15; lines 32–36

Although σ_1 to σ_4 cover all six SP test requirements (i.e., σ_0 covers <12, 32> and <32, 22>; σ_1 covers <22, 32> and <32, 12>; σ_2 covers <32, 12> and <12, 22>; σ_3 covers <22, 12> and <12, 32>), they do not detect the fault.

In contrast, the combinatorial SP coverage has combinatorial test requirements for every pair of SP requirements (e.g., $\{< 12, 22 >, < 22, 32 >\}$ and a test execution that covers this combinatorial requirement will detect the error.

4.3 CUVE framework

4.3.1 Overview

To make test executions cover targeted test requirements for concurrency coverage, CUVE generates thread schedule which is the execution order of multiple threads through a test execution. The CUVE framework has two major modules – *thread model analyzer* and *test generator*. The thread model analyzer monitors runtime information of the target program to construct an execution model, based on which CUVE estimates the feasible singular concurrency coverage requirements. The estimated feasible test requirements are transferred to the test generator.

To achieve high concurrency coverage, the test generator module repeatedly generates test executions by manipulating thread scheduling at runtime. CUVE has two testing phases–*Singular coverage based Testing Phase (STP)* and *Combinatorial coverage based Testing Phase (CTP)*. CUVE runs STP first and then CTP. This is because the singular coverage is "coarser" than the combinatorial coverage and, thus, targeting to achieve high singular coverage at early testing period generates more diverse thread schedules. When CUVE does not increase singular coverage for 10 consecutive test executions in STP (i.e., the singular coverage based test generation is saturated), CUVE advances to CTP and use the combinatorial concurrency coverage for the thread schedule generation.

4.3.2 Estimator of Feasible Test Requirement

To explain the dynamic analysis that estimate feasible test requirements of concurrency coverage metrics, I first define a *thread model* $\mathcal{M}$ of a multithreaded program as a finite set of threads, each of which consists of a finite sequence of atomic actions, where an action p has the following attributes:

- `thread(`p`)` is a thread executing p.
- `operator(`p`)` $\in$ `Sync` $\cup$ `Thread` $\cup$ `Data` indicates a type of p.
  - `Sync` = {`lock-hold`, `lock-acquire`, `unlock`}
  - `Thread` = {`thread-creation`, `thread-join`}
  - `Data` = {`read`, `write`}
- `operand(`p`)` indicates an operand of p.
  - For `operator(`p`)` $\in$ `Sync`, `operand(`p`)` is the corresponding lock.
  - For `operator(`p`)` $\in$ `Thread`, `operand(`p`)` is the corresponding thread.
  - For `operator(`p`)` $\in$ `Data`, `operand(`p`)` is the variable/memory location to read or write.
- `loc(`p`)` is the corresponding code location of p.

- `lockset(`p`)` is the set of locks held by `thread(`p`)` when p begins to execute.
- `next(`p`)` is the `lock-hold` actions of `thread(`p`)` that *first* accesses `operand(`p`)` after p.

I define the functions that relate two `lock-hold` actions p and p' of the same thread (i.e., `operator(`p`)=operator(`p'`)=lock-hold` and `thread(`p`)=thread(`p'`)`) as follows: [3]

- `lockset(`p`, `p'`)` is a set of locks that continuously guards p and p'.
- `next-lock(`p`)` is the `lock-hold` action of `thread(`p`)` that first holds `operand(`p`)` after p.
- `prev-lock(`p`)` is the `lock-hold` action of `thread(`p`)` that *most recently* holds `operand(`p`)` before p.
- `next-write(`p`)` is the `write` action of `thread(`p`)` that first writes `operand(`p`)` after p.
- `prev-lock(`p`)` is the `write` action of `thread(`p`)` that *most recently* writes `operand(`p`)` before p.

In addition, I define a *precedence relation* $\prec$ on the actions of $\mathcal{M}$ that represents ordering constraints between actions of two different threads t and t'. The ordering constraints are imposed at the time of thread creations.

> For any action p of thread t that occurs before t creates a new thread t' and for any action p' of t', $p \prec p'$.

I define an *interleaved execution* σ of a target multithreaded program as a sequence of actions of the all threads. During an interleaved execution σ, the program is at any program point with a state s. We introduce the following functions regarding σ and s:

- $\sigma[i]$ indicates ith action of σ.
- `enabled(`s`)` is a set of executable actions at s, through which s changes to another state s'.

Based on the thread model and the interleaved execution model, we define test requirements for the synchronization-pair (SP) coverage metric and the def-use (DU) coverage metric.

The definitions are as follows:

Definition 6. Synchronization-Pair (SP) Test Requirement
A pair of code locations ¡l_1, l_2¿ is an SP requirements, if the following conditions hold for σ:

1. *l_1 and l_2 have lock statements that are executed on the same lock m (i.e., there is σ such that* `loc(`$\sigma[i]$`)=`l_1, `loc(`$\sigma[j]$`)=`l_2, `operator(`$\sigma[i]$`)=operator(`$\sigma[j]$`)=lock-hold`,
   and `operand(`$\sigma[i]$`)=operand(`$\sigma[j]$`)=`m *for some $i < j$).*
2. *There is no* `lock-hold` *action on m between $\sigma[i]$ and $\sigma[j]$ (i.e., there is no k such that $i < k < j$,* `operand(`$\sigma[k]$`)=lock-hold`, *and* `operand(`$\sigma[k]$`)=`m*).*

Definition 7. Def-Use (DU) Test Requirement
A pair of code locations ¡l_1, l_2¿ is an DU requirements, if the following conditions hold for σ:

1. *l_1 is a write statement on a shared variable v and l_2 is a read or write statement on the same variable. (i.e., there is σ such that* `loc(`$\sigma[i]$`)=`l_1, `loc(`$\sigma[j]$`)=`l_2, `operator(`$\sigma[i]$`)=write`,
   `operator(`$\sigma[j]$`)=write` *or* `operator(`$\sigma[j]$`)=read`, *and* `operand(`$\sigma[i]$`)=operand(`$\sigma[j]$`)=`v *for some $i < j$).*

[3] `next-lock(`p`)`, `next-write(`p`)`, `prev-lock(`p`)` and `prev-write(`p`)` might be undefined when p is the last/first access of `operand(`p`)` by `thread(`p`)`.

2. *There is no* `write` *action on* v *between* $\sigma[i]$ *and* $\sigma[j]$ *(i.e., there is no* k *such that* $i < k < j$, `operand(`$\sigma[k]$`)=write`, *and* `operand(`$\sigma[k]$`)=`v*).*

Next, we discuss the conditions, under which SP and DU requirements can be covered during execution. We formally define the satisfaction criterion as follows:

Definition 8. SP Test Requirement Satisfaction Criteria
For an execution σ *of a program* P *and an SP requirement ¡*l_1, l_2*¿,* $\sigma \models < l_1, l_2 >$ *if there exist* i *and* j *(*$i < j$*) such that*

1. `loc(`$\sigma[i]$`)` $= l_1$ *and* `loc(`$\sigma[j]$`)` $= l_2$
2. `operator(`$\sigma[i]$`)=operator(`$\sigma[j]$`)=`*`lock-hold`*
3. `operand(`$\sigma[i]$`)=operand(`$\sigma[j]$`)`
4. *there is no* k *such that* $i < k < j$, `operator(`$\sigma[k]$`)=lock-hold`, *and* `operand(`$\sigma[k]$`)=operand(`$\sigma[i]$`)=operand(`$\sigma[j]$`)`

Definition 9. DU Test Requirement Satisfaction Criteria
For an execution σ *of a program* P *and an DU requirement ¡*l_1, l_2*¿,* $\sigma \models < l_1, l_2 >$ *if there exist* i *and* j *(*$i < j$*) such that*

1. `loc(`$\sigma[i]$`)` $= l_1$ *and* `loc(`$\sigma[j]$`)` $= l_2$
2. `operator(`$\sigma[i]$`)=write`
3. `operator(`$\sigma[j]$`)=write` *or* `operator(`$\sigma[j]$`)=read`
4. `operand(`$\sigma[i]$`)=operand(`$\sigma[j]$`)`
5. *there is no* k *such that* $i < k < j$, `operator(`$\sigma[k]$`)=write`, *and* `operand(`$\sigma[k]$`)=operand(`$\sigma[i]$`)=operand(`$\sigma[j]$`)`

The estimation phase computes and reports a set of SP and DU requirements that can be satisfied by possible thread interleavings. To do this, the technique builds a thread model $\mathcal{M}$ by executing a program once and collecting data such as actions and threads. $\mathcal{M}$ has a set of executed threads, where each thread has a sequence of actions, and `lock-hold` actions have their dynamic `lockset` information. From $\mathcal{M}$, the technique attempts to create every possible SP and DU test requirements as pairs of lock statements and data access statements. Then, the technique filters out some pairs that are definitely infeasible by checking (1) dynamic `lockset` relations, and (2) precedence relations. The pairs that are not filtered out (i.e., accepted) are reported as likely feasible.

I formally define the acceptance conditions of the SP requirements as follows. For SP requirement ¡`loc(`p`),loc(`q`)`¿ of `lock-hod` actions p and q on the same lock m, the technique accepts the pair when the following conditions hold:

- If `thread(`p`)=thread(`q`)`,

(SP1) q=`next-lock(`p`)`

- If `thread(`p`)`$\neq$`thread(`q`)`,

(SP2) `lockset(`p`, next-lock(`p`))` $\cap$ `lockset(`q`)` $= \emptyset$

(SP3) `lockset(`p`)` $\cap$ `lockset(prev-lock(`q`),` q`)` $= \emptyset$

(SP4) $q \not\prec p$

SP1 accepts consecutive `lock` statements executed from the same thread as a feasible pair. SP2, SP3, and SP4 define the conditions for the pairs from different threads. SP2 implies that p, `next(`p`)`, and q should *not* be protected by a common lock, so that q can execute consecutively after p (i.e., before `next(`p`)`). For example, suppose there exists m such that $m \in$ `lockset(`p`, next(`p`))` $\cap$ `lockset(`q`)`. Then, p and `next(`p`)` are continuously protected by m, and thus, q cannot execute consecutively after p. Hence, ¡`loc(`p`),loc(`q`)`¿ is filtered out. SP3 filters out infeasible conditions in a similar manner. SP4 filters out pairs that violates the precedence relation. If $q \prec p$, p cannot execute before q so that the corresponding pair is infeasible.

Similarly, I formally define the acceptance conditions of the SP requirements as follows. For DU requirement ¡`loc(`p`),loc(`q`)`¿, the technique accepts the pair when the following conditions hold:

- If `thread(`p`)`=`thread(`q`)`,

(DU1) q=`next-write(`p`)`

- If `thread(`p`)`$\neq$`thread(`q`)`,

(DU2) `lockset(`p`, next-write(`p`))` $\cap$ `lockset(`q`)` $= \emptyset$

(DU3) `lockset(`p`)` $\cap$ `lockset(prev-write(`q`), `q`)` $= \emptyset$

(DU4) $q \not\prec p$

4.3.3 Test generator

To generate an execution to cover target test requirements, CUVE controls the execution orders of read, write, and synchronization operations in a thread at runtime. Algorithm 1 describes how CUVE generates various test executions by controlling the execution orders of the multiple threads according to the STP or CTP thread schedule decision algorithms. These algorithms decide which thread to run at a time, based on the target test requirements.

The test generation algorithm (Algorithm 1) receives an initial state of a target program s_0, a testing phase flag *phase* (*phase* is STP or CTP), a set of estimated feasible singular coverage requirements $estimated_S$, and singular and combinatorial test requirements already covered in the previous test executions (i.e., $covered_S$, and $covered_C$). The algorithm initiates an execution by setting program state to s_0 (line 3).

To control thread execution order, CUVE suspends every enabled action p of a thread (line 5) by adding p to *paused* (line 7) if p synchronizes or accesses data (line 6). When all running threads are suspended (i.e., *paused*=`enabled(`s`)` at line 9), CUVE invokes a scheduling decision algorithm to select one action from *paused* to execute in the next step (lines 10–14). In other words, depending on the current testing phase (i.e., *phase*), CUVE executes either SingularDecision() (line 11), or CombinatorialDecision() (line 13) to select an action that may increase target coverage (Sections 4.3.4 and 4.3.5).

The test generation algorithm resumes the paused thread by executing the paused action selected (line 18). After the action is executed, CUVE determines a singular test requirement achieved by the action (SP requirement in lines 20–21 and DU requirement in lines 22–23). $last\_lock_s(l)$ returns the code location of the last lock action on the lock l at state s. $operator(p)$ returns a type of p and $operand(p)$ returns an operand of p (i.e., a lock variable for synchronization action and a data variable for read or write action). $loc(p)$ returns the code location of p. $last\_write_s(v)$ returns the code location of the last write

Input: s_0: an initial state
phase: the testing phase in a testing run, either *STP* or *CTP*
$estimated_S$: estimated feasible singular test requirements
$covered_S$: singular test requirements covered in a test
$covered_C$: combinatorial requirements covered in a test
Output: Updated $covered_S$ and $covered_C$

```
1  curr ← ∅; /* a set of singular coverage requirements covered in this execution.*/
2  paused ← ∅; /* a set of paused actions */
3  s ← s_0;
4  while enabled(s) ≠ ∅ do
5      p ← an action in enabled(s)\ paused;
6      if operator(p) is lock, read, or write then
7          Add p to paused ;
8      end
9      if paused = enabled(s) then
10         if phase = STP then
11             p ← SingularDecision(paused, s, curr, covered_S,estimated_S);
12         else if phase = CTP then
13             p ← CombinatorialDecision(paused, s, curr, covered_S, covered_C);
14         end
15         Remove p from paused ;
16     end
17     if p ∉ paused then
18         s ←execute(s, p); // updates s by executing p ;
19         c ← undefined; // singular coverage information
20         if operator(p) is lock then
21             c ← < last_lock_s(operand(p)), loc(p) > ;
22         else if operator(p) is read or write then
23             c ← < last_write_s(operand(p)), loc(p) > ;
24         end
25         if c ≠ undefined then
26             Add the elements of comb(curr, c) to covered_C ;
27             Add c to covered_S and curr ;
28         end
29     end
30 end
```

Algorithm 1: Test execution generation algorithm

action for a variable v at state s. Then, CUVE updates the achieved combinatorial concurrency coverage (line 26) and the achieved singular concurrency coverage (line 27) correspondingly. $comb(S, c)$ returns a set of sets $\{\{s_1, c\}, ...\{s_n, c\}\}$ where $S = \{s_1, ...s_n\}$ (for example $comb(\{1, 2\}, a) = \{\{1, a\}, \{2, a\}\}$). This process repeats until no enabled action remains (line 4), which corresponds to the program termination or a deadlock. [4]

[4]This test generation technique is inspired by the randomized scheduler in Sen [87]. The main difference is that CUVE decides an action to schedule using the coverage-based algorithms while Sen [87] decides randomly.

Input: *paused*: a set of paused actions
s: a current state
curr: singular requirements covered in this execution
$covered_S$: singular requirements covered in a test
$estimated_S$: estimated feasible singular test requirements
Output: An action *act* in *paused* to execute

SingularDecision(*paused*, s, *curr*, $covered_S$, $estimated_S$){
1 // The first rule
2 **if** $\exists p \in paused : cov_s(p) \notin covered_S$ **then**
3 | $act \leftarrow p$;
4 **else**
5 | // The second rule
6 | **if** $\exists p, q \in paused : cov_s(p, q) \notin covered_S$ **then**
7 | | $act \leftarrow p$;
8 | **else**
9 | | // The third rule
10 | | $act \leftarrow p \in paused$ such that $|\{t \in rel(estimated_S, loc(p)) \mid t \notin covered_S\}|$ is the smallest;
11 | **end**
12 **end**
13 **return** *act*; }

Algorithm 2: Scheduling decision algorithm for the singular concurrency coverage

4.3.4 Singular coverage based scheduler

Algorithm 2 is a scheduling decision algorithm (i.e., selecting an action from a set of paused actions to execute) to achieve high singular concurrency coverage fast. The algorithm receives a set of paused actions (*paused*), a current state (s), a set of singular test requirements covered in the current execution (*curr*), a set of singular test requirements covered in the previous test executions ($covered_S$), and a set of likely feasible singular test requirements ($estimated_S$). The algorithm uses several auxiliary functions. A function $cov_s(p)$ returns a singular test requirement that is covered if an action p is executed at a state s. A function $cov_s(p, q)$ returns a singular test requirement that is covered when q executes right after p executes at a state s.

The algorithm runs three rules to select an action to execute in order. The algorithm uses the first rule to select an action whose execution at a state s can immediately cover a new singular test requirement (i.e., a singular test requirement that has not been covered) (lines 1–3). If there are multiple choices of such actions, the algorithm arbitrary chooses one of those. If the first rule fails (i.e., if no paused action to increase coverage), the algorithm uses the second rule (lines 5–7) which selects an action p whose immediate subsequent action q can cover a new test requirement.

If the second rule fails, the algorithm uses the third rule which is heuristics to select an action of the least possibility to cover a new test requirement in later steps of the current test execution (lines 9–12). Specifically, for each action p in *paused*, the algorithm counts the number of the feasible test requirements which may be covered by p later (i.e., $< l_1, l_2 >$ may be covered by p later if $l_1 = loc(p)$ or $l_2 = loc(p)$). The third rule selects an action whose count is the smallest because such action has the least possibility to increase coverage (i.e., the other actions in *paused* have more possibility to increase coverage in later step), thus, least potential benefit to hold. $rel(estimated_S, loc(p))$ returns a subset of $estimated_S$ each of whose elements has $loc(p)$ as a component (for example, $< l_1, l_2 >$ has two components l_1 and l_2).

Input: *paused*: a set of paused actions
s: a current state
curr: singular requirements covered in this execution
$covered_S$: singular requirements covered in a test
$covered_C$: combinatorial requirements covered in a test
Output: An action *act* in *paused* to execute

CombinatorialDecision(*paused*, s, *curr*, $covered_S$, $covered_C$){
1 // The first rule
2 **if** $\exists p \in paused : comb(curr, cov_s(p)) \setminus covered_C \neq \emptyset$ **then**
3 | $act \leftarrow p \in paused$ such that $|comb(curr, cov_s(p)) \setminus covered_C|$ is the largest ;
4 **else**
5 | // The second rule
6 | **if** $\exists p, q \in paused : comb(curr, cov_s(p, q)) \setminus covered_C \neq \emptyset$ **then**
7 | | $act \leftarrow p \in paused$ such that $\exists q \in paused \setminus \{p\}$, $|comb(curr, cov_s(p, q)) \setminus covered_C|$ is the largest ;
8 | **else**
9 | | // The third rule
10 | | $act \leftarrow p \in paused$ such that $|\{t \in rel(covered_S, loc(p))|\ comb(curr, t) \setminus covered_C \neq \emptyset\}|$ is the smallest;
11 | **end**
12 **end**
13 **return** *act*; }

Algorithm 3: Scheduling decision algorithm for the combinatorial concurrency coverage

4.3.5 Combinatorial coverage based scheduler

Algorithm 3 is a scheduling decision algorithm of CTP to achieve high combinatorial concurrency coverage fast. It receives a set of paused actions (*paused*), a current state (s), a set of singular requirements covered in this execution (*curr*), a set of singular requirements and a set of combinatorial coverage requirements covered in the previous test executions ($covered_S$ and $covered_C$). The algorithm makes the scheduling decision according to the following three rules in order.

The first rule is to select an action that covers the largest number of new combinatorial requirements (lines 1–4). A set of combinatorial coverage requirements achieved by an action p includes combinations of the set of singular requirements covered in this execution (*curr*) and the singular requirement covered by p at s ($cov_s(p)$). Thus, the number of new combinatorial coverage requirements covered can be computed by counting the combinatorial requirements in $comb(curr, cov_s(p))$ that are not in $covered_C$. If there are multiple choices of such actions, the algorithm arbitrary chooses one of those.

The second rule selects an action whose immediate subsequent action can cover the largest number of new combinatorial test requirements in the next turn (lines 6–9). The algorithm first finds two different actions p and q in *paused* that can cover a new singular requirement $cov_s(p, q)$. For such p and q, the algorithm checks if covering $cov_s(p, q)$ achieves new combinatorial requirements (line 7).

The third heuristic rule selects an action that can cover the smallest number of new combinatorial requirements in later steps in this execution (lines 11–14). The algorithm selects an action p such that $rel(covered_S, loc(p))$ has the smallest number of the covered singular coverage requirements that compose new combinatorial requirements which will be covered later in a running execution. CUVE selects such p as such action has the least possibility to increase coverage soon, thus, least potential benefit to hold.

4.4 Experiment design

4.4.1 Research questions

We designed a series of experiments to evaluate how much CUVE improves the effectiveness and efficiency of concurrent program testing in terms of concurrency coverage achievement (RQ1 and RQ2) and fault detection (RQ3 and RQ4) compared to the conventional concurrent program testing techniques. Also, we investigated the impact of the combinatorial concurrency coverage based test generation feature (i.e., CTP) of CUVE with regard to the coverage achievement effectiveness and fault detection effectiveness (RQ5).

RQ1: Coverage achievement effectiveness. To what extent does CUVE achieve higher coverage than the other conventional testing techniques do in the combinatorial concurrency coverage?

RQ2: Coverage achievement efficiency. How fast does CUVE achieve certain levels of the combinatorial concurrency coverage, compared to the other conventional testing techniques?

RQ3: Fault detection effectiveness. To what extent does CUVE detect more faults than the other conventional testing techniques?

RQ4: Fault detection efficiency. How fast does CUVE detect a fault compared to the other conventional testing techniques?

RQ5: Impact of CTP of CUVE on coverage achievement and fault detection. To what extent does CUVE achieve higher coverage and detect more faults than the CUVE without CTP (i.e., with only STP)?

To answer these research questions, we studied 19 different test generation techniques (Section 4.4.2) on 65 mutated faulty versions of three Java programs and six real-fault objects used in related literature (Section 4.4.3).

4.4.2 Test generation techniques

The independent variable of the experiment is the technique used for concurrent program test generation. We used 12 random noise-injection techniques, a randomized scheduling technique, a systematic test generation technique, and three bug pattern-directed scheduling techniques as conventional concurrent program test generation techniques to compare with CUVE. Also, to answer RQ5, we used a variant of CUVE that uses only STP without CTP.

Random noise-injection techniques. We used 12 versions of a random noise-injection technique [27, 51]. These techniques insert artificial time delay before synchronization operations and read-/write operations that may access shared variables to induce diverse thread scheduling in repeated test executions. We constructed 12 different techniques by controlling the maximum amount of random delay at a target operation as 5, 10, and 15 milliseconds and controlling the probability of a noise-injection per target operation as 10%, 20%, 30%, and 40%.

Randomized scheduling technique. We used a randomized scheduling technique presented by Sen [87]. This technique suspends running threads before every synchronization operation and data access, and then randomly selects and resumes one thread at a time.

Systematic testing technique. We used Java PathFinder core (JPF-core) version 7 as a systematic testing technique. We used JPF with the depth-first search strategy and the maximum depth

bound 10^6. We configured JPF to terminate when JPF finds a violation (i.e., an uncaught exception) to measure time for detecting a fault. We used JPF only for study on fault detection ability, not for coverage achievement because modifying JPF to measure combinatorial concurrency coverage would require heavy extra effort.

Bug pattern-directed testing techniques. We used RaceFuzzer [88], AtomFuzzer [75], and DeadlockFuzzer [49], which generate thread schedules of specific patterns based on bug prediction information targeting data race bugs, atomicity violations, and cyclic deadlock bugs, respectively. [5] We operated each technique to obtain fault prediction information prior to test generation, and then fed the information to generate test executions. Since these tools originally terminated when it had tried test generation with all fault prediction information, we re-executed the test generations for 1000 seconds of testing time in the experiments. We used these bug pattern-directed techniques only for study on fault detection ability, not for on coverage achievement because these tools do not measure combinatorial concurrency coverage and we used them off the shelf.

CUVE-c. To assess the impact of the combinatorial concurrency coverage based test generation feature, we used a variant of CUVE that does not use CTP but STP only. [6]

4.4.3 Study objects

We applied the techniques to two groups of objects. First, to study the testing techniques with regard to various concurrency faults, we used mutation objects each of which has systematically mutated faulty versions. Second, to study the testing technique with regard to real faults, we used real-fault objects that have been used for concurrent program testing research.

Mutation objects

As mutation objects, we used three Java programs `ArrayList`, `HashMap`, and `TreeSet` in Java Util library (see Table 4.1). We selected these programs because they are often used as benchmark programs in study on testing techniques for concurrent programs [46,48,75,88]. In addition, these programs contain many synchronization blocks from which a large number of mutants can be generated. Each program is used with the synchronization wrapper to provide thread-safe behaviors. For example, `ArrayList` is always wrapped with `Collections.SynchronizedList` in the test cases used for our study. We fixed all known faults of the programs before the experiments.

Like other concurrent program testing studies [46, 49, 75, 88, 88], we set a test case (test driver) to initialize shared data structures of a target program and then create multiple threads each of which runs a corresponding method with fixed input values and terminates. We used the same test case for the faulty mutants too.

Each mutant of the mutation object is generated by seeding one fault to an original target program. We used both *synchronization mutation operators* [9, 36] and *expression mutation operators* [3, 25]. [7] Synchronization mutation operators transform a synchronization operator (e.g., `synchronized` block) of

[5] We chose these three tools because they are the only publicly available tools for Java programs. We could not use PECAN [46] whose result was often different from its description. We reported this issue to the PECAN developers.

[6] CUVE-c is similar to the thread scheduling algorithm of Hong et al. [41] except that CUVE-c utilizes both SP and DU coverages while Hong *et al.* [41] uses SP only.

[7] We used the following synchronization mutation operators: exchange synchronized block parameters, remove synchronized block, shrink synchronized block, and split synchronized block. Also we used the following expression mutation operators: access flag change, argument order change, arithmetic operator change, logical connector change, and relational operator change (Table 4.2).

Table 4.1: Study objects used for the CUVE experiments

Type	Program	Size (LOC)	Number of threads	Number of faulty versions
Mutation objects	ArrayList	3090	27	18 (4, 179)
	HashMap	3941	27	12 (19, 169)
	TreeSet	4049	22	35 (26, 190)
Real fault objects	Airlines	487	9	1
	Crawler	1281	17	1
	Log4j-509	16425	3	1
	Log4j-1507	16301	3	1
	Pool-146	5735	3	1
	Pool-184	4718	3	1

a target program [9]. An expression mutation operator changes one expression of a target program such as an arithmetic operator or a method call [25]. We applied expression mutation operators as well as synchronization mutation operators because expression mutation operators can induce a fault which can be detected only at specific thread scheduling.

We removed mutants whose faults were not detected by any testing technique in 1000 seconds (i.e., likely equivalent mutants). We also removed mutants whose faults were detected by all testing techniques in less than one second. The fifth column of Table 4.1 shows the number of mutants used in the experiments. The numbers in the parenthesis are the number of (likely) equivalent mutants and the number of mutants killed in less than one second by all techniques.

Real-fault objects

As real-fault objects, we chose six real-world Java applications with already known real faults (Table 4.1). We collected these programs from an existing testing benchmark [25] and the study objects used for related literature [45, 71]. Among many candidates, we selected the six programs whose testing results vary depending on the testing techniques (i.e., the testing results of these objects are useful to compare the different techniques).

The first two objects `Airlines` and `Crawler` are the study objects used in Hrubá et al. [45]. `Airlines` is a flight schedule simulator. It has a high-level data race and violates the assert statement. `Crawler` is a web crawling system, and it has a data race bug that leads to null pointer dereference errors. To run these objects, we used the original test cases provided in their distributions. The other five objects and their test cases are obtained from Apache open-source projects. `Log4j-509` is a Java logging system (Apache Commons Log4j ver. 1.0) and it causes null pointer dereference errors due to unsynchronized consequent accesses to a shared variable (i.e., atomicity violation (reported as Issue# 509)). `Log4j-1507` is a later version of Log4j (ver. 1.1) that has a data race bug (Issue# 1507) and causes null pointer dereference errors. `Pool-146` is a later version of Pool (ver. 1.5) that causes deadlock at wait operation (Issue# 146). `Pool-184` is another version of Pool (ver 1.5.5) where a race condition causes livelocks (i.e., infinite loop). The first four of the five real-fault objects are compiled in the SIR benchmark [25]. `Logj4j-509`, `Logj4j-1507`, `Pool-146`, and `Pool-184` are used as study objects in the literature [71].

Table 4.2: Mutation operators used for the study

Category	Mutation operator description
Synchro-	Exchange synchronized block parameters
nization	Remove synchronized block
mutation	Shrink synchronized block
operator	Split synchronized block
Expression	Access flag change
mutation	Argument order change
operator	Arithmetic operator change
	Logical connector change
	Relational operator change

4.4.4 Testing runs and measurement

In one testing run, we applied a testing technique to one version of a mutation object or one real-fault object for 1000 seconds (our preliminary experiment results did not change much after 1000 seconds in most cases. See Figure 4.3 and 4.4 for example). We performed 30 testing runs per every version of a mutation object and per a technique to obtain statistical reliability in the result [82]. Similarly, we performed 30 testing runs per a real-fault object. We consider that a test execution detects a fault if a target object raises an uncaught exception, violates assertions, or exceeds given time bounds (i.e., to detect deadlock/livelock). For each pair of a technique and a study object, we measured the coverage achievement and the fault detection ability as follows:

Coverage achievement. For a mutation object, we measured the average numbers of the combinatorial test requirements covered over the 30 testing runs for the correct version of the object by each testing technique. Similarly, we measured the average numbers of the combinatorial test requirements covered over the 30 testing runs for a real-fault object by each testing technique.

Fault detection ability. For a mutation object, the fault detection ability $F_x(p,t)$ of a testing technique x on a target program p at time t in one testing run is defined as follows:

$$F_x(p,t) = \frac{\Sigma_{m \in \mathcal{M}(p)} f_x(m,t)}{|\mathcal{M}(p)|}$$

where $\mathcal{M}(p)$ is a set of mutated faulty versions of p, $f_x(m,t) \in \{0,1\}$ indicates if the fault of a mutated faulty version m has been detected by the time instant t (1 if detected, 0 otherwise) in a testing run. For example, suppose that the average fault detection ability $F_{cuve}(\texttt{ArrayList}, 100)$ is 0.94 (see Figure 4.4) where $\mathcal{M}(\texttt{ArrayList})$=$\{m_1,...,m_{18}\}$, $f_{CUVE}(m_1, 100) = 1$,..., f_{CUVE} $(m_{18}, 100) = 0$ for 18 faulty mutants m_1 to m_{18} of `ArrayList`. This means that the probability that CUVE detects a fault in `ArrayList` after testing `ArrayList` for 100 seconds is 94%. We measure the average fault detection ability of a testing technique over the 30 testing runs on each mutation object.

For a real-fault object p, we measure the average fault detection ability by using $f_x(p,t)$ over 30 testing runs.

Table 4.3: Coverage achievements of the testing techniques

Program	RN-5ms			RN-10ms			RN-15ms			RS	CUVE
	min	avg	max	min	avg	max	min	avg	max		
ArrayList	82874.7	91632.9	101041.0	80782.6	90269.7	100808.9	78835.4	89051.4	100902.6	75945.1	**117030.1**
HashMap	58895.3	74667.7	92323.5	56313.2	73923.6	92117.8	56364.0	72978.4	91677.6	49691.7	**98785.4**
TreeSet	98223.7	122941.8	149261.9	96508.3	120977.7	144711.3	95313.4	118603.2	141005.5	83186.9	**215772.1**
Airlines	14266.1	14354.1	14486.3	14093.0	14256.9	14470.8	14006.6	14186.3	14442.7	14354.1	**14572.3**
Crawler	28296.4	29221.1	**30326.3**	28366.2	29240.0	30170.0	28352.1	29064.1	29879.3	28247.7	30105.7
Log4j-509	12956.8	13102.6	13251.5	13100.5	13192.8	**13317.0**	13083.4	13141.9	13253.6	13239.3	13257.0
Log4j-1507	3494.4	3528.6	**3540.0**	3529.9	3537.5	**3540.0**	3532.4	3538.1	**3540.0**	**3540.0**	**3540.0**
Pool-146	40349.3	40640.7	41104.5	40350.0	40679.2	41168.8	40368.8	40617.8	41190.5	39949.6	**41215.1**
Pool-184	63310.9	68993.0	75046.0	55677.3	66547.9	71861.1	51550.9	64621.4	73093.6	**75925.4**	74562.8

4.4.5 Tool implementation

CUVE has 78 Java classes (15 KLOC). CUVE is compiled and executed with Java 7 (1.7.0_40). To monitor and control thread scheduling of an execution at runtime, CUVE inserts probes using Soot [103] before and after every synchronization actions (e.g., monitor enter/exit, wait/notify, thread create/join) and every data access actions (e.g., read/write) that may access to a shared variable in bytecode level.

4.4.6 Threat to validity

External. The conclusion of the study might be different for other objects. However, we believe that the conclusion is still valid for other programs that are interesting for the concurrent program testing research because the study objects in our study have been widely used for evaluating concurrent program testing techniques. In addition, testing runs are generated using one test case/input, as the most concurrent program testing techniques do. With multiple test cases, however, the conclusion will be still valid if intensive testing for each test case is necessary, which is often the case for testing concurrent programs whose unintended interleaving behaviors trigger errors.

Internal. To avoid incorrect results due to a bug in the CUVE implementation, we have tested the CUVE implementation intensively. Also, although we implemented the random noise-injection techniques and the randomized scheduling technique faithful to the original descriptions (corresponding tools for Java were not available), our implementation may not be equivalent to the original techniques.

Construct. The mutation operators used for the study may not generate all types of concurrent program faults. However, we used as many mutation operators as possible for Java so to generate many faulty versions. In addition, we conducted the experiment with real-fault objects to alleviate this limitation.

4.5 Experiment results

4.5.1 RQ1. coverage achievement effectiveness

Table 4.3 shows the average coverage achievements of the studied techniques over 30 testing runs. The first column shows the study object name. The second to the four columns show the minimum, the average, and the maximum coverages achieved after 1000 seconds by the four random noise-injection techniques (RN) with the delay probability 10% to 40% and the maximum random delay of 5 milliseconds (RN-5ms). The fifth to the tenth columns show the coverage results of RN-10ms and RN-15ms similarly.

Table 4.4: Time to reach certain level of coverage achievement (in seconds)

Program	70%				80%				90%				100%			
	RN_W	RN_B	RS	CUVE	RN_W	RN_B	RS	CUVE	RN_W	RN_B	RS	CUVE	RN_W	RN_B	RS	CUVE
ArrayList	n/a	58.8	n/a	**24.7**	n/a	306.3	n/a	**57.2**	n/a	n/a	n/a	**133.4**	n/a	n/a	n/a	**772.9**
HashMap	n/a	35.3	n/a	**6.3**	n/a	99.9	n/a	**12.2**	n/a	422.3	n/a	**39.6**	n/a	n/a	n/a	**407.6**
TreeSet	n/a	n/a	n/a	**138.4**	n/a	n/a	n/a	**226.9**	n/a	n/a	n/a	**426.4**	n/a	n/a	n/a	**995.5**
Airlines	33.2	3.4	**0.4**	1.0	38.5	3.7	**0.4**	1.1	126.6	14.7	**1.7**	4.8	n/a	n/a	n/a	**997.2**
Crawler	3.7	**0.5**	1.0	1.6	7.0	**0.5**	2.9	1.9	77.8	**2.4**	24.5	7.7	n/a	**993.1**	n/a	n/a
Log4j-509	1.9	0.3	**0.2**	0.6	3.5	**0.4**	**0.4**	0.6	7.8	0.8	**0.5**	0.8	n/a	**949.0**	n/a	n/a
Log4j-1507	0.7	**0.1**	**0.1**	0.2	0.8	**0.1**	**0.1**	0.2	44.8	0.9	6.3	**0.8**	n/a	**28.3**	501.2	380.8
Pool-146	10.8	0.5	**0.1**	10.6	11.5	10.0	**0.1**	31.4	31.2	**10.4**	11.7	72.6	n/a	n/a	n/a	**977.0**
Pool-184	n/a	21.0	**10.5**	11.8	n/a	40.3	12.9	12.2	n/a	95.0	**26.1**	53.3	n/a	n/a	**975.0**	n/a

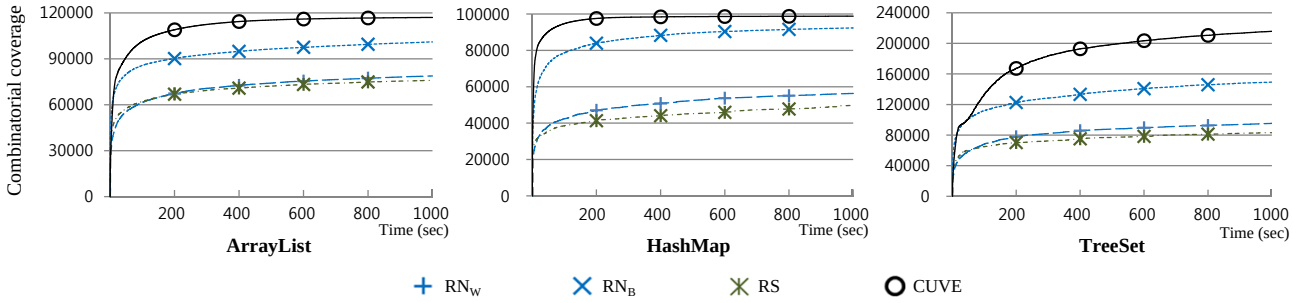


Figure 4.3: Coverage achievement over time per testing technique

The 11th column describes the result of the randomized scheduling technique (RS). The last column shows the results of CUVE. The highest coverage per object is marked in a bold font.

The coverage of CUVE is up to 53.0% higher (= (215772.1-141005.5)/141005.5 on `TreeSet` compared with the maximum coverage of the RN-15ms techniques) or equal to all maximum coverages of the RN-5ms, the RN-10ms, and the RN-15ms techniques for all objects except `Crawler` and `Log4J-509`; the maximum coverage of the RN-5ms techniques is 0.7% higher than CUVE for `Crawler` and the maximum coverage of the RN-10ms techniques is 0.5% higher than CUVE for `Log4j-509`. Compared with RS, CUVE achieves up to 159.4% higher coverage (= (215772.1-83186.9)/83186.9 on `TreeSet`) than RS for all objects except `Pool-184` for which the coverage of RS is 1.8% higher than CUVE.

4.5.2 RQ2. coverage achievement efficiency

Table 4.4 shows how much time each technique spends on average over 30 testing runs to achieve 70%, 80%, 90%, and 100% of the highest combinatorial concurrency coverage observed in the experiment. For example, the highest combinatorial coverage of `ArrayList` is 117030.1 achieved by CUVE (see the second row of Table 4.3). The columns RN_W and RN_B indicate the worst (i.e., the longest time taken) and the best (i.e., the shortest time taken) results of the 12 RNs to reach certain levels, respectively. 'n/a' indicates that a testing technique did not reach the coverage level in 1000 seconds. The least time to reach a certain coverage level per object is marked in a bold font.

For the three mutation objects, CUVE achieves all levels of coverage faster than the 12 RNs and RS. For example, CUVE reaches 70% coverage level in 6.3 seconds for `HashMap`, which is 5.6 (=35.3/6.3) times faster than RN_B which achieves 70% coverage level in 35.3 seconds. Figure 4.3 illustrates this

Table 4.5: Fault detection abilities of the testing techniques

Program	RN-5ms			RN-10ms			RN-15ms			RS	JPF	RF	AF	DF	CUVE
	min	avg	max	min	avg	max	min	avg	max						
ArrayList	0.58	0.70	0.84	0.59	0.74	0.95	0.58	0.72	0.91	0.55	0.44	0.68	0.00	0.00	1.00
HashMap	0.42	0.64	0.88	0.39	0.64	0.92	0.38	0.63	0.91	0.34	0.17	0.58	0.00	0.00	0.92
TreeSet	0.75	0.81	0.87	0.75	0.81	0.87	0.75	0.81	0.87	0.62	0.17	0.25	0.00	0.00	0.94
Airlines	0.75	0.83	0.94	0.31	0.66	1.00	0.19	0.53	0.88	1.00	1.00	1.00	0.00	0.00	1.00
Crawler	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	0.91	0.00	1.00	1.00	0.00	1.00
Log4j-509	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	0.00	0.00	1.00
Log4j-1507	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	0.00	0.00	1.00
Pool-146	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	0.00	1.00
Pool-184	0.87	0.97	1.00	0.57	0.89	1.00	0.37	0.83	1.00	1.00	1.00	1.00	0.00	0.00	1.00

superior efficiency of CUVE on the three mutation objects clearly; the CUVE line is always above the 12 RNs and RS lines on all three mutation objects during the entire 1000 seconds.

For the six real-fault objects, CUVE increases coverage slower than RN_B during the early period of testing, but continues to increase coverage even after the coverage of RN_B is saturated and eventually achieves higher coverage than RN_B on the three real-fault objects. Similarly, CUVE increases coverage slower than RS during the early testing period, but continues to increase coverage after the coverage of RS is saturated and eventually achieves higher coverage than RS for all real-fault objects except `Pool-184`.

In other words, CUVE continues to generate various distinct test executions to satisfy explicit thread scheduling goals to cover diverse combinatorial test requirements for relatively long time while the 12 RN techniques and RS generate redundant test executions after the early test period.

4.5.3 RQ3. fault detection effectiveness

Table 4.5 shows the average fault detection abilities of the studied techniques over 30 testing runs. The second to the forth columns show the minimum, the average, and the maximum fault detection abilities of the four RN-5ms after testing a target program for 1000 seconds (the fifth to the tenth columns are similar). The 11th to the 15th columns describe the fault detection abilities of RS, a systematic testing technique (JPF), the three bug pattern-directed techniques (RandomFuzzer (RF), AtomFuzzer (AF), and DeadlockFuzzer (DF)), respectively. The last column is for CUVE.

Table 4.5 shows that CUVE achieves the highest fault detection abilities (i.e., 1.00 indicating that CUVE always detects a fault) on all studied objects except `HashMap` and `TreeSet`. For `HashMap`, CUVE still achieves higher or equal fault detection abilities (i.e., 0.92 indicating that CUVE detects a fault in `HashMap` with the probability 0.92 over all mutants/faults and over 30 testing runs) compared to all the other techniques. For example, CUVE achieves 2.7 (=0.92/0.34), 5.4 (=0.92/0.17), and 1.6 (=0.92/0.58) times higher fault ability for `HashMap` compared to RS, JPF, and RF, respectively. For `TreeSet`, CUVE achieves higher fault detection abilities (i.e., 0.94) than all the other techniques.

For the real-fault objects, CUVE consistently achieves the highest fault detection abilities (i.e., 1.00), whereas the fault detection abilities of the other techniques except RF varies for different objects/faults. For example, all RN techniques except the max RN-10 ms sometimes fail to detect a fault in `Airlines` (i.e., fault ability is less than 1.00). RF shows high fault detection (i.e., 1.00) for all real-fault objects, even when an object has no data race bugs. This is because RF predicts 2–18 data race bugs for every real-fault object. We guess that RF could detect the faults of the other types because data race predictions nearby the actual fault may lead RF to generate thread schedules useful to detect the actual

Table 4.6: Time to reach certain level of fault detection ability (in seconds)

Program	70%						80%						90%						100%					
	RN_W	RN_B	RS	JPF	RF	CUVE	RN_W	RN_B	RS	JPF	RF	CUVE	RN_W	RN_B	RS	JPF	RF	CUVE	RN_W	RN_B	RS	JPF	RF	CUVE
ArrayList	n/a	24.0	n/a	n/a	n/a	7.6	n/a	65.9	n/a	n/a	n/a	13.0	n/a	511.7	n/a	n/a	n/a	40.6	n/a	n/a	n/a	n/a	n/a	169.2
HashMap	n/a	69.7	n/a	n/a	n/a	3.4	n/a	121.2	n/a	n/a	n/a	8.9	n/a	221.7	n/a	n/a	n/a	22.7	n/a	748.1	n/a	n/a	n/a	301.4
TreeSet	340.9	25.1	n/a	n/a	n/a	22.8	n/a	68.8	n/a	n/a	n/a	70.3	n/a	511.9	n/a	n/a	n/a	117.7	n/a	n/a	n/a	n/a	n/a	763.3
Airlines	n/a	536.5	50.2	90.0	5.7	44.0	n/a	605.9	63.3	181.0	8.3	45.0	n/a	662.2	100.6	200.0	9.0	74.7	n/a	977.1	154.9	253.0	22.7	76.1
Crawler	40.7	2.7	375.2	n/a	9.4	1.9	208.0	2.8	655.0	n/a	10.2	1.9	436.6	3.5	843.1	n/a	10.5	2.5	642.7	10.7	n/a	n/a	37.6	4.5
Log4j-509	4.0	0.4	0.4	0.5	0.9	1.0	5.3	0.5	0.5	0.5	1.1	1.2	7.2	0.8	0.6	0.5	15.3	1.4	16.0	1.0	1.3	0.5	374.7	2.0
Log4j-1507	49.2	3.0	29.6	0.5	1.0	0.9	82.6	5.0	35.1	0.5	1.9	0.9	109.1	6.6	46.5	0.5	3.0	0.9	254.2	14.8	96.5	0.5	4.2	1.3
Pool-146	15.0	10.8	10.6	0.5	11.2	10.9	16.8	11.0	11.6	0.5	11.5	11.3	17.4	13.0	11.7	0.5	12.8	11.8	25.4	14.1	12.4	0.5	17.0	13.3
Pool-184	n/a	37.0	20.5	0.5	53.4	12.0	n/a	42.3	24.1	0.5	59.0	12.1	n/a	57.7	27.9	0.5	95.8	12.3	n/a	97.8	32.8	0.5	159.1	12.6

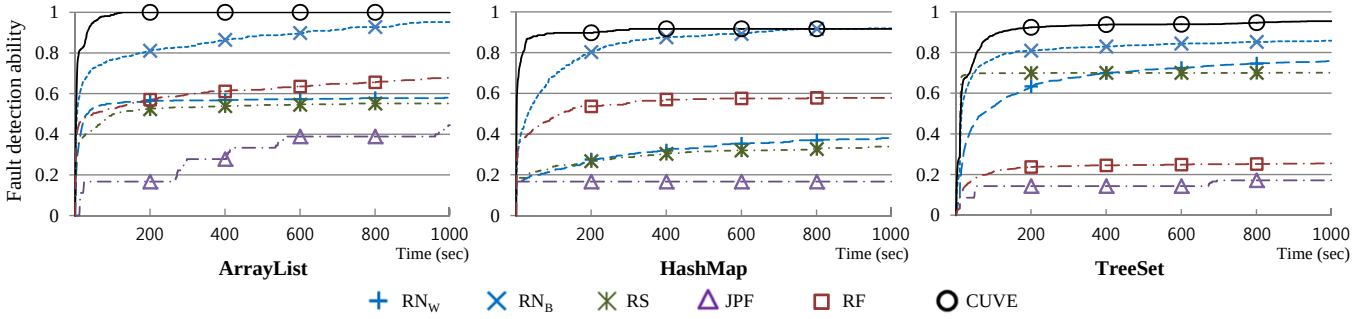


Figure 4.4: Fault detection abilities over time per testing technique

fault.

Note that DF completely miss faults in all objects including `Pool-146` which has a deadlock fault. Similarly, AF completely miss faults in all objects (except `Crawler` and `Pool-146`) including `Log4j-509` which has an atomicity violation fault. We guess that AF and DF may generate imprecise fault prediction information due to complex target program structure and focus to generate improper (i.e., correct) test executions instead of the erroneous ones.

4.5.4 RQ4. fault detection efficiency

Table 4.6 shows how much time each technique spends on average over 30 testing runs to achieve 70%, 80%, 90%, and 100% of the highest fault detection ability observed in the experiment. For example, for `HashMap`, CUVE reaches the 70% fault detection ability level (i.e., achieves fault detection ability 0.65 (=0.92×70%)) in 3.4 seconds on average. The columns RN_W and RN_B indicate the worst and the best results of the 12 RNs to reach certain levels, respectively. 'n/a' indicates that a testing technique did not reach the fault detection ability level in 1000 seconds. We omit AF and DF from Table 4.6 because these techniques failed to reach the lowest fault detection ability level (i.e., 70%) on most objects in 1000 seconds.

For the three mutation objects, CUVE achieves all fault detection ability levels faster than all the other techniques. For example, CUVE reaches the 70% level in 3.4 seconds for `HashMap`, which is 20.5 (=69.7/3.4) times faster than RN_B which achieves 70% level in 69.7 seconds. Figure 4.4 illustrates this superior fault detection efficiency of CUVE on the three mutation objects clearly; the CUVE line is always above the lines of the other techniques on all three mutation objects except `TreeSet` where the RS line is slightly above the CUVE line before 50 seconds.

Table 4.7: Comparison between CUVE-c and CUVE on coverage achievement and fault detection ability

Program	Coverage		Fault detection	
	CUVE-c	CUVE	CUVE-c	CUVE
ArrayList	109786.2	**117030.1**	0.88	**1.00**
HashMap	**98844.1**	98785.4	0.90	**0.92**
TreeSet	116146.8	**215772.1**	0.70	**0.94**
Airlines	14554.6	**14572.3**	**1.00**	**1.00**
Crawler	29713.7	**30105.7**	**1.00**	**1.00**
Log4j-509	13256.0	**13257.0**	**1.00**	**1.00**
Log4j-1507	**3540.0**	**3540.0**	**1.00**	**1.00**
Pool-146	38582.9	**41215.1**	**1.00**	**1.00**
Pool-184	71686.6	**74562.8**	**1.00**	**1.00**

For the six real-fault objects, compared to RN_B and RS, CUVE is constantly faster (up to 13.5 and 344.7 times faster than RN_B and RS on `Airlines` and `Crawler` to reach the 80% level respectively) to reach all levels of fault detection ability on all objects except `Log4j-509` and `Pool-146`. On `Pool-146`, CUVE is faster than RN_B to reach 90% and 100%. Compared to RF, CUVE is constantly faster (up to 12.6 times faster on `Pool-184` to reach the 100% level) to reach all levels of fault detection ability on all objects except `Airlines` and `Log4j-509`. On `Log4j-509`, CUVE is faster than RF to reach 90% and 100%. JPF is faster to reach all fault detection ability levels than CUVE on all objects except `Airlines` and `Crawler`. [8] We guess that JPF detects real faults in the study fast because the test cases for the real-fault objects were created by the original developers to manifest the faults for debugging/bug-report purpose. Thus, JPF can detect such faults quickly without exploring the search space deeply. Note that all test cases of the real-fault objects use only three threads except the test cases of `Airlines` and `Crawler` which uses 9 and 17 threads respectively (Table 4.1) and JPF is slower than CUVE for these two objects (JPF completely fail to detect a fault in `Crawler` that uses 17 threads).

4.5.5 RQ5. impact of CTP on CUVE performance

Table 4.7 shows that CUVE achieves up to 85.8% (=(215772.1-116146.8)/116146.8 on `TreeSet`) higher or equal combinatorial concurrency coverage compared to CUVE-c for all objects except `HashMap`, for which CUVE-c achieves 0.1% higher coverage than CUVE. [9] In addition, Table 4.7 shows that CUVE achieves up to 34% (=(0.94-0.70)/0.70 on `TreeSet`) higher or equal fault detection ability compared to CUVE-c for all objects. Therefore, we can conclude that the combinatorial concurrency coverage can be an effective method to improve both coverage achievement and fault detection ability of CUVE.

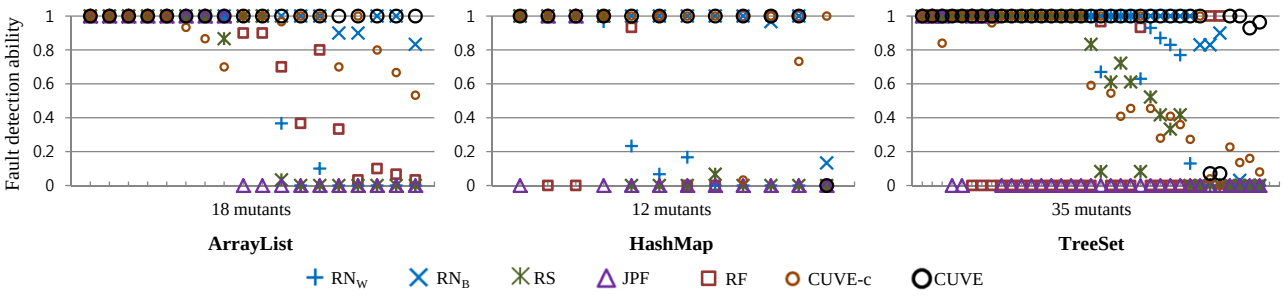


Figure 4.5: Fault detection abilities of the testing techniques per mutant

4.6 Discussion

4.6.1 High effectiveness of CUVE for various faults

We have generated diverse mutants to study the effectiveness of the testing techniques with regard to various types of faults (Section 4.4.3). Figure 4.5 shows fault detection abilities of the studied testing techniques per mutant. Figure 4.5 shows that the big circles representing the fault detection abilities of CUVE are on top of the graphs (i.e., 1.00 meaning that every testing run (out of total 30 testing runs) detects a fault) for all three objects except for the 12th one out of the 12 mutants of `HashMap` and the four (the 30th, the 31th, the 34th, and the 35th mutants) out of the 35 mutants of `TreeSet`. This indicates that CUVE detects all types of faults under study completely on `ArrayList` and most completely on `HashMap` and `TreeSet`.

In contrast, the other techniques are not effective to detect specific types of faults (see the symbols representing the other techniques at the bottom/middle of the figure). For example, all techniques almost completely miss the 32nd mutant/fault of `TreeSet` while CUVE detects the fault completely. In addition, the figure shows that CUVE has higher fault detection ability than CUVE-c which sometimes fail to generate fully diverse executions that can reveal the faults.

Furthermore, Table 4.5 shows that CUVE detects all faults of the real-fault objects completely while the other techniques miss specific faults frequently (e.g., faults in `Airlines` and `Pool-184`). These observations imply that CUVE is a general technique to detect various concurrency faults more effectively and more consistently than the other techniques.

4.6.2 Benefits of the combinatorial concurrency coverage

Figure 4.6 shows singular coverage achievement on `TreeSet` using several testing techniques including CUVE and CUVE-c. The figure shows that CUVE achieves higher singular coverage than CUVE-c after 80 seconds, which indicates that the combinatorial coverage-based thread scheduling algorithm (Algorithm 3) can improve singular coverage achievement more than the singular coverage-based thread scheduling algorithm (Algorithm 2). This is because CTP continues to generate various test executions to satisfy many diverse combinatorial test requirements explicitly, which enables CUVE to escape the pitfall of local optimum of STP that targets to cover singular test requirements in a greedy manner. Thus, we can expect that CTP will improve the fault detection ability because of the strong correlation between the singular concurrency coverage and fault detection ability [43]. Thus, these observations can

[8]Since JPF reports execution time only in seconds, we write 0.5 when JPF reports an execution time as zero second.

[9]All differences between CUVE and CUVE-c on coverage achievement (except `Log4j-509`) and fault detection ability are statistically significant (Wilcoxen test with α=0.05).

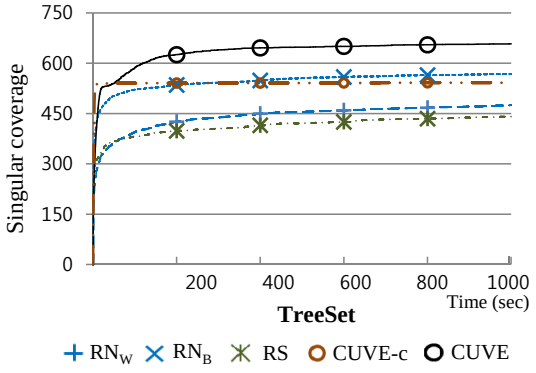


Figure 4.6: Singular coverage achievement over time

another evidence that the combinatorial concurrency coverage can be an effective tool to improve fault detection.

4.6.3 Comparison with Maple

The most similar technique to CUVE is Maple [120] which tests multithreaded C/C++ programs. Similar to CUVE, Maple utilizes concurrency coverage that defines test requirements over two to four statements of data access and synchronization actions. Although direct empirical comparison between CUVE and Maple is not feasible due to different target programming languages, we conjecture that CUVE has advantages over Maple for the following two reasons:

First, CUVE generates thread schedules to achieve as many new combinatorial requirements as possible for every test execution (Section 4.2) while Maple targets one test requirement for one test execution. Thus, we expect that CUVE is faster to detect a fault than Maple. Second, a test requirement used for Maple is defined for interleaving between two threads while the combinatorial requirements for CUVE can represent interleavings of more than two threads (for example, see the general race bug in Section 4.2.2). Thus, we expect that the combinatorial coverage can be used to generate more diverse test executions than the coverage metric by Maple.

4.7 Summary of this chapter

In this chapter, I present the combinatorial concurrency coverage and a new concurrent program test generation technique CUVE that utilizes the combinatorial concurrency coverage to improve testing effectiveness and efficiency. The experiment result shows that CUVE is more effective and efficient forachieving coverage, and also for detecting faults than the other conventional techniques studied. In addition, through the experiments, I show that the combinatorial concurrency coverage is more useful than the existing concurrency coverage metrics for CUVE to achieve high fault detections.

Chapter 5. Regression Testing Using Concurrency Coverage Metric

5.1 Introduction

Many software developers today write multithreaded programs to make their software to utilize multi-core processors for high performance. Multithreaded programs, like all other types of software, evolve through code changes as their requirement changes over time. In addition, a code change is often made to reduce synchronization overhead and increase the number of concurrent threads to improve performance. A challenge in revising code of multithreaded programs is that seemingly insignificant code change in multithreaded code may easily introduce new *concurrency bugs* if the code change affects interactions among threads. Thus, developers must carefully validate each change of a multithreaded program.

Unfortunately, developers often fail to recognize subtle and adverse changes that introduce concurrency bugs in evolving multithreaded programs. A concurrency bug is difficult to detect because the failure may or may not appear depending not only on program input, but also non-deterministic thread schedule. A recent study on evolving software found that concurrency bugs is the most popular type of long-living bugs introduced by adverse program changes (i.e., called *dormant bugs* which survive through many revisions) [18]. Moreover, concurrency bugs are difficult to fix correctly. Studies on the patches in evolving software show that patches for concurrency bugs are often not complete or introduce new bugs due to side-effects [58, 117]. For this reason, Lu *et al.* [58] claims that concurrency bug fixes are non-trivial in practice and developers need new techniques to ensure the correctness of multithreaded program changes.

For last two decades, many regression testing techniques have been proposed for sequential (single-threaded) programs and become standard testing methods in practice [118]. Regression testing focuses the changed part of an evolving software to detect new bugs introduced by the change effectively and efficiently. A regression testing technique selects/generates test inputs that are expected to explore the changed part of the target program. To identify such test inputs, regression testing techniques analyze the target program code and the test coverage results.

To detect adverse program changes of multithreaded programs (i.e., regression bugs), regression testing should exercise various thread schedules that explore different concurrency behaviors related to the code changes. In contrast to the sequential program domain, there are only few regression testing techniques for multithreaded programs (see Section 5.2); most regression testing techniques do not consider thread schedule as a part of test cases. Moreover, for effective regression testing, a regression testing technique for multithreaded programs should precisely identify the thread schedules that exercise changed behaviors of a target multithreaded program, and then generate specific thread schedules to exercise the changed behaviors.

In this chapter, I present a new regression testing technique *Recurve* for multithreaded programs which utilizes *combinatorial concurrency (CC) coverage metric* to systematically explore different thread schedules related to the changed code. A key idea of Recurve is to identify the changed program behaviors

due to code change in terms of the test requirements of the concurrency coverage metric, and then generate thread schedules toward achieving more test requirements. In other words, as Recurve covers more test requirements, it will detect adverse program change with higher probability. The experiment result confirms that Recurve detects regression bugs with higher probability faster than the other related techniques for the study objects. In summary, the contribution of the new regression testing technique is the following:

- I present a new thread scheduling technique that effectively tests code changes in multithreaded programs.
- I define an effective coverage metric for regression testing of multithreaded programs (i.e., CC).
- The carefully designed empirical study demonstrates that Recurve is more effective and efficient to find concurrency bugs introduced by program changes, compared to the existing testing techniques for multithreaded programs.

5.2 Existing approaches

5.2.1 Static and dynamic analyses for finding regression bugs

Several researchers have investigated regression bugs in evolving multithreaded programs, and improved the existing dynamic and static analysis techniques to efficiently detect/predict concurrency bugs in evolving multithreaded programs.

Deng *et al.* [23] proposes to utilize dynamic/static analysis techniques for detecting specific types of concurrency bugs (e.g., data race, atomicity violation) to identify program behaviors changes in evolving multithreaded programs. Sadowski and Yi [83] reports the cases of predicting data race bugs over multiple program versions of multithreaded programs, and suggest to utilize data race prediction history on the previous versions for better concurrency bug prediction.

RECONTEST [100] is a dynamic bug prediction technique that reports concurrency bugs introduced by a program change. In particular, RECONTEST utilizes AssetFuzzer [52] for detecting a set of suspicious interleaving patterns (i.e., atomic-set bugs) as bugs, and then selects the regression bugs based on the program changes in sequential logics as well as synchronizations.

SimRT [121] extends a dynamic data race detection/testing technique RaceFuzzer [88] to support efficient data race detections in evolving multithreaded programs. SimRT analyzes a code change in two versions of a multithreaded program, and generates a multithreaded test driver for RaceFuzzer, which is likely to reveal data race bugs caused by the program change. SimRT uses the thread scheduler of RaceFuzzer off-the-shelf. Since RaceFuzzer generates specific thread schedules targeted for given data race information, SimRT may not be effective for detecting different types of concurrency bugs in a regression testing (see Section 5.5.1).

Since the aforementioned techniques rely on specific bug prediction patterns/techniques, these techniques are limited for detecting particular types of concurrency bugs, rather than generating effective and efficient thread schedules to detect broad range of concurrency bugs in regression testing.

5.2.2 Thread schedule generation for regression testing

CAPP [47] is a thread scheduling technique to explore new behaviors caused by program change. CAPP utilizes a search strategy to guide a model-checker to explore the program states impacted by

a code change prior to the other states. The key idea of CAPP is that a preempted interleaving (i.e., making a context-switch from a current thread to another thread) at the changed code is more likely to explore new behavior of the modified program. To generate such interleavings in model-checking, CAPP analyzes the program code changes and defines certain statements related to the changes as "impacted". Then, CAPP assigns high priorities to state transitions caused by preemption at the impacted statements/locations.

There are two heuristics to assign high priority (i.e., prioritization mode): `SOME` and `ALL`. The `SOME` option assigns a high priority to every context-switch where one thread is at the impacted location. The CAPP with the `ALL` option assigns a higher priority to a context-switch if all concurrent threads are at the impacted locations (i.e., a context-switch occurs from the impacted location to another impacted location). In addition, CAPP has the seven heuristics to identify the impacted locations: `CLASS`, `METHOD`, `LINE`, `CLASS-ON-STACK`, `METHOD-ON-STACK`, `LINE-ON-STACK`, and `FIELD`. The `CLASS/METHOD/LINE` heuristics consider all code elements of a Java class/method/line as impacted if at least one code element in the target class/method/line is changed. [1] The `FIELD` heuristic considers a code element is impacted if the code line accesses a variable/field whose declaration is changed. The `CLASS-ON-STACK/METHOD-ON-STACK/LINE-ON-STACK` heuristic considers a current location in an execution is impacted if any element in the call stack as an impacted location with respect to the `CLASS/METHOD/ LINE` heuristic.

However, CAPP may not be effective for regression testing because injecting a preemption at the impacted locations may not cover a new behavior if a current thread and the other concurrent threads do not access the same data structure at the time. Moreover, a model-checking technique with CAPP has a limited scalability with respect to a target program size, due to the state-explosion problem of underlying model-checker such as JPF [105]. To the best of the authors' knowledge, CAPP is the only thread scheduling technique to explore new behaviors of a modified program.

5.2.3 Coverage-guided testing of multithreaded programs

Coverage-guided testing techniques for multithreaded programs utilizes concurrency coverage metrics to enumerate various thread interaction cases, and generate tests to achieve high coverage for effective and efficient testing of multithreaded programs. A recent empirical study on the concurrency coverage metrics shows that increasing concurrency coverage achievement in a tests is highly correlated with improving fault detections of multithreaded programs [44].

Wang *et al.* [109] presents a coverage-based search strategy for systematic testing technique. The search strategy utilizes a concurrency coverage metric PSet to enumerates different cases of interleavings in a target program, and intends the state exploration to cover more new interleaving cases. Hong *et al.* [41] presents a dynamic thread scheduling algorithm that achieves high concurrency coverage fast. Hong *et al.* utilizes a dynamic analysis to estimate feasible test requirements and targets only likely feasible test requirements for efficient testing. Similarly, Maple [120] utilizes a dynamic analysis to obtain the test targets which are expected to be achievable, and then generates a specific thread schedule for achieving each test requirement.

However, there is no application of coverage-guided testing techniques to effective regression testing of multithreaded programs to date. A similar yet different application with the regression testing is memoization of thread scheduling [22,120]. These techniques aim to support better utilization of multiple

[1] If a synchronized block is changed, CAPP considers all statements inside the block as changed.

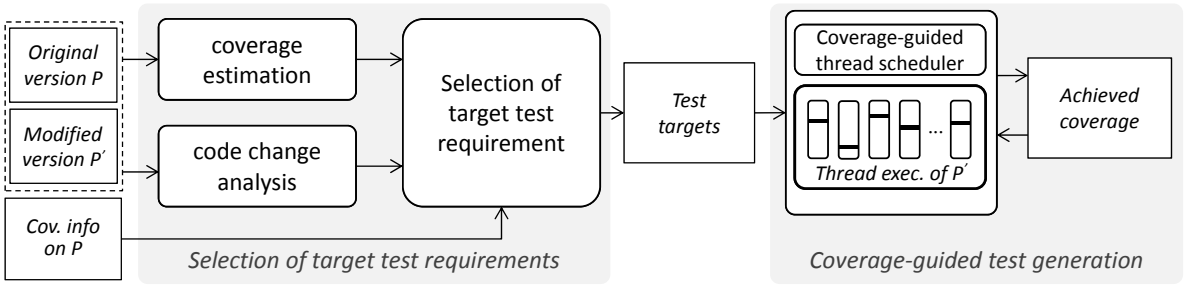


Figure 5.1: Overall process of Recurve

program inputs for testing a single version of a target program. However, they are not applicable for regression testing where the target program code is changed.

5.3 Recurve: a coverage based regression testing technique

I present *Recurve* which is a coverage based regression testing technique for multithreaded programs. For the original version and a modified version of a target program, Recurve selectively generates thread schedules to explore the new concurrent behaviors of the modified version. Particularly, Recurve compares the test requirements of the Combinatorial Concurrency (CC) metric (Section 5.3.2) for the original and the modified versions and then Recurve controls thread schedules targeted for achieving the new test requirements for the modified version because the new test requirements can capture the new behaviors effectively.

5.3.1 Overview

Figure 5.1 describes the overall process of Recurve. There are two sub-processes: test target selections and coverage-guided test generation. In the test target selection process, Recurve selects the test requirements of the CC metric that are likely feasible and relevant to the new behavior of the modified version as *test targets*. For the given original version P and the modified version P' of a target program, Recurve first estimates the feasible test requirements on P' and prioritizes the estimated test requirements in order of the relevance to the new behavior of P' (see Section 5.3.3).

In the coverage-guided test generation process, Recurve generates a sequence of test executions based on the test targets and the test requirements achieved in the testing so far. To test the modified version with various thread schedules, Recurve repeatedly executes P' while controlling thread schedules to cover as many test targets as possible. In particular, the thread scheduler of Recurve dynamically decides an execution order of operations in multiple threads based on the runtime status of P' and the target test requirements covered so far.

5.3.2 Combinatorial Concurrency (CC) coverage metric

The Combinatorial Concurrency (CC) metric is a concurrency coverage metric that generate various test requirements of different execution orders in synchronization, data accesses, and their combinations for a multithreaded program. The CC metric basically defines two types of test requirements: *singular test requirements* and *combinatorial test requirements*.

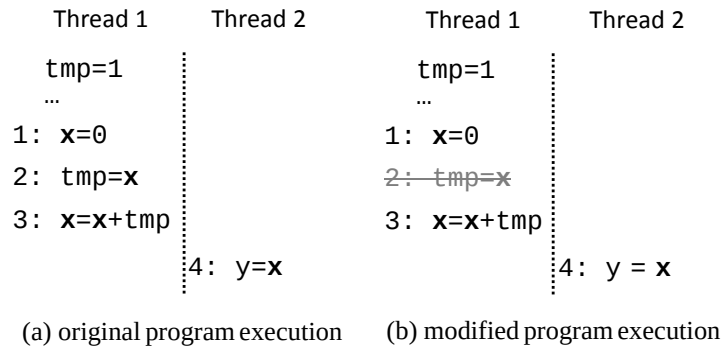


Figure 5.2: Examples of multithreaded programs and executions

A singular test requirement is defined as an ordered pair of two code locations whose statements access the same variable or hold the same lock. A singular test requirement for the two statements p and q that access the same shared variable v is denoted as $(loc(p), loc(q))$. For two statements that access the same shared variable v, an execution σ covers singular test requirement $(loc(p), loc(q))$ if p executes before q, p and q access v, and no statement that reads or writes v executes between p and q. [2]. Similarly, for two statements that hold the same lock m, an execution σ covers singular test requirement $(loc(p), loc(q))$ if p executes before q, p and q access m, and no statement holds m between p and q.

The singular test requirement of the CC metric is a generic form of pairwise concurrency coverage metrics [44] such as Sync-Pair [41] and PSet [119]. The CC metric subsumes Sync-Pair by definition. In addition, the singular test requirement of the CC metric subsumes the concurrency coverage metrics defined over shared variable accesses such as PSet [119] and Def-Use [98] because the CC metric consider all kinds of two adjacent data accesses on the same shared variable. PSet and Def-Use do not define a test requirement for a pair of operations in the same thread or a pair of two reads on the same shared variable. However, in the regression testing context, these pairs are useful to capture changed concurrent behavior (explained later with Figure 5.2).

A *combinatorial test requirement* is defined as an ordered pair of two singular test requirements. For two singular test requirements $(loc(p_0), loc(q_0))$ and $(loc(p_1), loc(q_1))$, a combinatorial test requirement is denoted as $\langle (loc(p_0), loc(q_0)), (loc(p_1), loc(q_1)) \rangle$. The satisfaction relation of a combinatorial test requirement by an execution σ is defined as follows:

$\sigma \models \langle (loc(p_0), loc(q_0)), (loc(p_1), loc(q_1)) \rangle$ if

1. $\sigma \models (loc(p_0), loc(q_0))$, and
2. $\sigma \models (loc(p_1), loc(q_1))$, and
3. $\exists\ i,\ j : i < j \wedge \sigma[i] = q_0 \wedge \sigma[j] = q_1$ where $\sigma[i]$ indicates (i-1)-th operation of σ

In other words, a combinatorial test requirement $\langle (loc(p_0), loc(q_0)), (loc(p_1), loc(q_1)) \rangle$ is satisfied by an execution if the execution satisfies the two singular test requirements $(loc(p_0), loc(q_0))$ and $(loc(p_1), loc(q_1))$ and q_0 precedes q_1 in the execution. The main motivation for constructing the combinatorial test requirements is the pairs/combinations of test requirements of a coverage metric can capture more diverse behaviors of a target program than the set of singular test requirements.

Figure 5.2 explains why the CC metric is more effective to capture the changed concurrent behavior than the other concurrency coverage metrics. Figure 5.2 shows two versions of a program and their

[2] I assume sequentially consistent memory model as our technique is not aid for detecting sequential consistency violations with weak memory models.

Table 5.1: Test requirements covered in the example executions

Coverage metric		Test req. covered in Figure 5.2(a)	Test req. covered in Figure 5.2(b)
PSet		(3,4)	(3,4)
Def-Use		(1,3),(2,3),(3,4)	(1,3),(3,4)
CC	Singular TR	(1,2),(2,3),(3,4)	(1,3),(3,4)
	Comb. TR	⟨(1,2),(2,3)⟩ ⟨(1,2),(3,4)⟩ ⟨(2,3),(3,4)⟩	⟨(1,3), (3,4)⟩

interleaved executions. The program has two threads Thread1 and Thread2 and two shared variables `x` and `y`. In addition, Thread1 has a local variable `tmp`. The program change from the original version (Figure 5.2(a)) to the modified version (Figure 5.2(b)) is that `tmp=x` at Line 2 is removed. This code change affects the execution result as `y` values of the original version and the modified version are 0 and 1 respectively.

Note that an effective concurrency coverage metric *cm* for regression testing should have test requirements that can guide a regression testing technique to generate new executions of the modified version that do not exist for the original version (i.e., *cm* should have at least one test requirement that is covered by Figure 5.2(b) but not by Figure 5.2(a)).

Table 5.1 shows the covered test requirements of PSet, Def-Use and the CC for the two executions in Figure 5.2. For Figure 5.2, PSet does not have any useful test requirement that can guide to generate different executions caused by the program change since both executions cover the same test requirements (i.e., (3,4)) (i.e., PSet is not an effective coverage metric for regression testing of multithreaded programs). This deficiency of PSet is mainly because PSet does not consider a case of data accesses within the same thread. Def-Use does not have a useful test requirement either as the set of test requirements covered by the modified version is a subset of those covered by the original version. In contrast, CC has a new singular test requirement (1,3) which is covered by the modified version but not covered by the original version. The combinatorial test requirement of the CC metric is even more useful because a combinatorial test requirement ⟨(1,3),(3,4)⟩ precisely captures the new execution of the modified version. This example demonstrates that the CC metric is more effective for regression testing of multithreaded programs than PSet and Def-Use because test requirements of CC can capture new multithreaded behaviors of the modified version effectively.

5.3.3 Selection of test targets

Recurve selects test targets as the test requirements of the CC metric on P' that are likely to be covered (i.e., feasible). [3] Recurve determines whether or not a test requirement on P' is feasible based on the result of a dynamic analysis on P', the source code of P and P', and the coverage measured in a test of P. Recurve first selects likely feasible singular test requirements, then selects combinatorial test requirements each of which consists of two likely feasible singular test requirements. Recurve selects

[3] As a sequential program may have an unreachable branch and it is a undecidable problem to identify unreachable branches in a target program, some test requirement of a concurrency coverage metric may be infeasible and it is difficult to check if a given test requirement is feasible [92].

target singular test requirements in the following three steps:

Obtaining test requirements

Recurve first obtains the following three sets of the test requirements on P' based on the dynamic analysis on P', the coverage information on P and the source code of P and P' [4] :

- C_P: a set of test requirements on P' that correspond to the covered test requirements on P. Recurve considers that a test requirement r' of P' corresponds to a test requirement r of P if all code lines of r' correspond to those of r.
- E_P: a set of test requirements on P' that correspond to the estimated test requirement on P by the dynamic analysis.
- $E_{P'}$: a set of test requirements on P' estimated as feasible to cover by the dynamic analysis

Recurve obtains the correspondence between the code lines of P and P' by textual comparison. In particular, Recurve computes the longest common sequence of code lines for every pair of source code files in the P and P'. [5] Two code lines in different versions are corresponding if their texts are identical and they appear at the same index of the longest common sequence. The dynamic analysis used by Recurve is an extension of coverage estimation algorithm proposed in Hong *et al.* [41] by adding analyses on data access information.

Selecting target test requirements

Recurve selects the target test requirements that belong to $E_{P'}$ or C_P but do not belong to $E_P \setminus C_P$ (i.e., $(E_{P'} \cup C_P) \setminus (E_P \setminus C_P)$) which are marked grey in Figure 5.3. [6] This selection is based on the following conjectures:

1. A test requirement $r' \in E_{P'}$ is likely covered because the dynamic analysis concludes so. Also, a test requirement $r'' \in C_P$ is likely covered as the corresponding test requirement r on P was actually covered. This is because the logics/structures of P and P' are similar to each other.
2. A test requirement r' on P' that corresponds to r on P that was estimated as feasible to cover but not actually covered (i.e., $E_P \setminus C_P$) is likely infeasible to cover.

Prioritizing selected test requirements

To detect regression errors fast, Recurve classifies the test targets into the following three classes according to the relevance to the new behaviors of P':

Class A: $E_{P'} \setminus (E_P \cup C_P)$
Class B: $E_{P'} \cap C_P$
Class C: $C_P \setminus E_{P'}$

Class A is more related to the new behaviors of P' than Class B and Class C because Class B and Class C capture the behaviors of P' that correspond to the old behaviors of P. Note that every test requirement on the new code lines in P' belongs to $E_{P'} \setminus$Class B. *Class B* is more related to the new

[4] I assume that the coverage achievement in a test of P is given.
[5] Recurve utilize the `diff` utility of Linux.
[6] $A \setminus B$ indicates a subset of A that does not overlap with B.

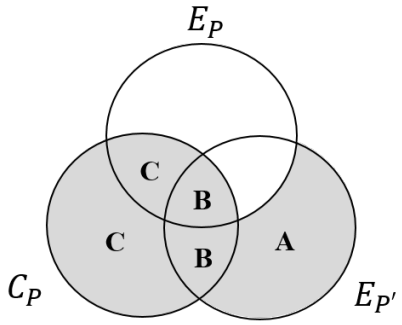


Figure 5.3: Selected test requirements among C_P, E_P and $E_{P'}$

behaviors of P' than Class C since the dynamic analysis does not consider Class C as feasible to cover (i.e., Class C $\cap$ $E_{P'} = \emptyset$). Thus, Recurve tries to cover the test targets in Class A, Class B, and Class C in order.

5.3.4 Coverage guided test generation

Recurve generates test executions by repeatedly executing P' with a *coverage-guided thread scheduler*. As shown in Figure 5.1, the thread scheduler runs in a middle of test executions to decide ordering of thread executions at runtime. Based on the test targets (i.e., target test requirements to cover) and the coverage achieved by the previous executions, the thread scheduler generates thread scheduling decisions to lead the execution to cover not-yet-covered test targets. Recurve targets the test requirements of Class A, Class B, and Class C in order (Section 5.3.3). Recurve changes a class of test targets if the coverage increase during the last few test executions is below a given threshold.

Recurve uses a test generation algorithm which extends the one in Hong *et al.* [41] in the following two ways:

1. Recurve utilizes the CC metric which subsumes the Sync-Pair metric used in [41].
2. It operates in two modes targeting to cover two different types of test requirements (i.e., targeting singular test requirement phase (STP) or targeting combinatorial test requirement phase (CTP)).

Algorithm 4 describes how the thread scheduler controls the execution orders of read, write, and synchronization operations of threads by deciding which thread to run at runtime. Algorithm 4 receives the following inputs: an initial state of a target program s_0, a testing phase flag *phase* (either STP or CTP), a set of singular target test requirements $target_S$, a set of combinatorial target test requirements $target_C$ a set of singular test requirements covered in the previous test executions $covered_S$, and a set of combinatorial test requirements covered in the previous test executions $covered_C$.

Recurve invoke the algorithm with an initial state s_0, $phase$ = STP, $target_S$ = ClassA, $target_C$ = ClassA $\times$ ClassA, $covered_S = covered_C = \emptyset$. The algorithm initializes a current program state s as s_0 (Line 3). To control thread execution order, Recurve suspends every enabled operation p of threads at s (i.e., $p \in$`enabled(`s`)`) (Line 5) by adding p to *paused* (Line 7) if p synchronizes or accesses data (Line 6). [7] When all running threads are suspended (i.e., *paused*=`enabled(`s`)` at Line 9), the algorithm invokes SingularDecision() at Line 11 (Section 32), or CombinatorialDecision() at Line 13 (Section 13) to select an operation from *paused* to execute in the next step (Lines 10–14). Then, the algorithm resumes the paused thread by executing the selected operation (Line 18). After the operation is executed, the

[7] We denote system dependent auxiliary functions in a typewriter font such as `enabled()` and `execute()`.

Input: s_0: an initial program state
phase: the testing phase in a testing run, either *STP* or *CTP*
$target_S$: a set of singular test targets
$target_C$: a set of combinatorial test targets
$covered_S$: a set of singular test requir. covered so far
$covered_C$: a set of combinatorial test requir. covered so far
Output: Updated $covered_S$, $covered_C$, $target_S$, $target_C$

```
1  curr ← ∅; /* a set of singular requirements covered in this execution.*/
2  paused ← ∅; /* a set of paused operations */
3  s ← s0; /* a current state */
4  while enabled(s) ≠ ∅ do
5      p ← an operation in enabled(s)\ paused;
6      if operator(p) is lock, read, or write then
7          Add p to paused ;
8      end
9      if paused = enabled(s) then
10         if phase = STP then
11             p ← SingularDecision(paused, s, covered_S,target_S);
12         else if phase = CTP then
13             p ← CombinatorialDecision(paused, s, curr, covered_S, covered_C, target_C);
14         end
15         Remove p from paused ;
16     end
17     if p ∉ paused then
18         s ←execute(s, p); // updates s by executing p ;
19         c ← undefined; // singular coverage information
20         if operator(p) is lock then
21             c ← (last_lock_s(operand(p)), loc(p)) ;
22         else if operator(p) is read or write then
23             c ← (last_acc_s(operand(p)), loc(p)) ;
24         end
25         if c ≠ undefined then
26             Add comb(curr, c) to covered_C ;
27             Remove comb(curr, c) from target_C ;
28             Add c to covered_S and curr ;
29             Remove c from target_S ;
30         end
31     end
32 end
```

Algorithm 4: Test execution generation algorithm

algorithm identifies a new covered singular requirement c (Lines 20–23). $operator(p)$ and returns a type of p. $last\_lock_s(l)$ returns the code location of the last lock operation on the lock l at state s. $operand(p)$ returns an operand of p (i.e., a lock variable for synchronization operation and a data variable for read or write operation). $loc(p)$ returns the code location of p. $last\_acc_s(v)$ returns the code location of the last write or read operation for a variable v at state s. Then, Recurve updates the covered combinatorial test requirements (Line 26) and the combinatorial test targets (Line 27). $comb(S, c)$ is a set of combinatorial

Input: *paused*: a set of paused operations
s: a current program state
$covered_S$: a set of singular test requir. covered so far
$target_S$: a set of singular test targets
Output: An operation *op* in *paused* to execute

```
   SingularDecision{
 1 // The first rule
 2 if ∃p ∈ paused : cov_s(p) ∈ target_S ∨ cov_s(p) ∉ covered_S then
 3 |   op ← p;
 4 else
 5 |   // The second rule
 6 |   if ∃p, q ∈ paused : cov_s(p, q) ∈ target_S ∨ cov_s(p, q) ∉ covered_S then
 7 |   |   op ← p;
 8 |   else
 9 |   |   // The third rule
10 |   |   op ← p ∈ paused such that |rel(target_S, loc(p))| is the smallest;
11 |   end
12 end
13 return op; }
```

Algorithm 5: Scheduling decision algorithm based on the singular coverage

test requirements obtained by combining a set of singular test requirements S with c (i.e., $S \times \{c\}$) (for example $comb(\{a, b\}, c) = \{\langle a, c\rangle, \langle b, c\rangle\}$). Similarly, the achieved singular coverage and *curr* (Line 28) and the singular test targets (Line 29) are updated. This process repeats until no enabled operation remains (Line 4), which corresponds to the program termination or a deadlock. [8]

Singular coverage based scheduler

Algorithm 5 selects an operation from a set of paused operations to execute to achieve high singular concurrency coverage fast. $cov_s(p, q)$ returns a singular test requirement that is covered when q executes right after p at a state s.

The algorithm applies the following three rules to select an operation to execute in order:

1. The first rule selects an operation at a state s that will immediately cover a singular test target or uncovered singular test requirement (Lines 1–3). [9] If there are multiple such operations, the algorithm arbitrarily chooses one of them. If the first rule fails (i.e., no paused operation to increase target coverage immediately), the algorithm uses the second rule.

2. The second rule (Lines 5–7) selects an operation p whose immediate subsequent operation q will cover a uncovered singular test target. If the second rule fails, the algorithm uses the third rule.

3. The third rule selects to release an operation that has least potential benefit to hold (i.e., most unlikely to cover a uncovered singular test target later) (Lines 9–12). For each operation p in *paused*, the algorithm counts the number of the test targets which may be covered by p later (i.e., (l_1,l_2) may be

[8] This algorithm description is inspired by Sen [87].

[9] The first and the second rules allow to select an operation that covers an already covered test target since there can be many different executions covering the test target some of which may reveal a regression error.

Input: *paused*: a set of paused operations
s: a current program state
curr: a set of singular requirements covered in this execution
$covered_S$: a set of singular test requir. covered so far
$covered_C$: a set of combinatorial test requir. covered so far
$target_C$: a set of combinatorial test targets
Output: An operation *op* in *paused* to execute

CombinatorialDecision{
1 // The first rule
2 **if** $\exists p \in paused : comb(curr, cov_s(p)) \cap target_C \neq \emptyset \ \vee \ comb(curr, cov_s(p)) \setminus covered_C \neq \emptyset$ **then**
3 $op \leftarrow p \in paused$ such that $|(comb(curr, cov_s(p)) \cap target_C) \cup (comb(curr, cov_s(p)) \setminus covered_C)|$ is the largest ;
4 **else**
5 // The second rule
6 **if** $\exists p, q \in paused : comb(curr, cov_s(p, q)) \cap target_C \neq \emptyset \vee comb(curr, cov_s(p, q)) \setminus covered_C \neq \emptyset$ **then**
7 $op \leftarrow p \in paused$ such that $\exists q \in paused \setminus \{p\}$, $|(comb(curr, cov_s(p, q)) \cap target_C) \cup (comb(curr, cov_s(p, q)) \setminus covered_C)|$ is the largest ;
8 **else**
9 // The third rule
10 $op \leftarrow p \in paused$ such that $|\{t \in rel(covered_S, loc(p))|\ comb(curr, t) \cap target_C \neq \emptyset\}|$ is the smallest;
11 **end**
12 **end**
13 **return** *op*; }

Algorithm 6: Scheduling decision algorithm based on the combinatorial coverage

covered by p later if $l_1 = loc(p)$ or $l_2 = loc(p)$). The third rule selects an operation whose count is the smallest because such operation has the least possibility to cover uncovered test targets (i.e., the other operations in *paused* have more possibility to cover uncovered test targets later). $rel(target_S, loc(p))$ returns a subset of $target_S$ each of whose elements has $loc(p)$ as a component (for example, (l_1,l_2) has two components l_1 and l_2).

Combinatorial coverage based scheduler

Algorithm 6 selects an operation from a set of paused operations to execute to achieve high combinatorial concurrency coverage fast. The algorithm applies the following three rules to select an operation to execute in order:

1. The algorithm selects an operation that will immediately cover the largest number of combinatorial test requirements which are either test targets or not-yet-covered test requirements (Lines 1–3). If there are multiple such operations, the algorithm arbitrary chooses one of them. If the first rule fails, the algorithm uses the second rule.

2. The second rule selects an operation whose immediate subsequent operation will cover the largest number of combinatorial test requirements which are either test targets or not-yet-covered test requirements (Lines 4–7). If the second rule fails, the algorithm uses the third rule.

Table 5.2: Study objects used for the Recurve experiments

Type	Program	# of faulty versions	Size (LOC)	# of changed lines	# of threads
Mutation object	ArrayList	18 (4,179)	3090	1.33	27
	HashMap	12 (19,169)	3941	1.42	27
	TreeSet	35 (26,190)	4049	1.29	22
Real fault object	Groovy	1	361	60	3
	Lang	1	990	3	3
	Pool-107	1	1693	148	3
	Pool-120	1	1614	201	3
	Pool-146	1	5735	29	3

3. The third rule selects an operation that can cover the smallest number of combinatorial test targets later in this execution (Lines 8–10) since the operation has least potential benefit to hold.

5.4 Experiment design

I have empirically evaluated the effectiveness and the efficiency of Recurve for detecting regression faults in multithreaded programs. I conducted a series of test generations with Recurve and six testing techniques including CAPP [47], and measured fault detections in the generated tests. These experiments use real faulty programs and mutated programs (i.e., programs containing various synthetic faults) as target programs to examine if a technique can detect various types of regression faults. The experiment results demonstrate that Recurve is more effective and more efficient for detecting regression faults for most cases in the experiments.

5.4.1 Research questions

I have studied the following two research questions:

- **RQ1 (fault detection effectiveness)**: To what extend does Recurve detect regression faults in multithreaded programs, compared to the other testing techniques?
- **RQ2 (fault detection efficiency)**: How fast is Recurve to detect a regression fault, compared to the other test generation techniques?

To answer these questions, we applied total 29 test generation techniques of six types with 70 regression faults in 8 target Java programs. For each technique and a target program, we if the technique detects the faults, and how much time is spent for generating the failing executions.

5.4.2 Target programs

I generated total 65 faulty versions of 3 programs for the mutation objects. I used three Java library programs `ArrayList`, `HashMap`, and `TreeSet` as the original programs and then generate multiple faulty versions systematically (i.e., through a mutation tool) as the modified versions. I selected these programs because they are often used as study objects in concurrent program testing research [46, 48, 75, 88]. In

addition, these three programs contain many synchronization blocks from which mutants with various concurrent behaviors can be generated. [10] Like other concurrent program testing studies [46,49,75,88,88], we set a test case (test driver) to initialize shared data structures of a target program and then create multiple threads each of which runs a corresponding method with fixed input values and terminates.

A mutant of a target program is generated by seeding one fault to an original target program. I used both *synchronization mutation operators* [9, 36] and *expression mutation operators* [3, 25]. I used a mutation tool Sofya [1] and applied our own implementation of synchronization mutation operators. [11].

Each mutant is generated by applying a unique mutation operator for a target statement/expression. Synchronization mutation operators transform a synchronization statement (e.g., `synchronized` block) of a target program [9]. For example, the remove synchronized block mutation operator [9] moves the statements in a synchronized block to the outer code block, and deletes the synchronized block statement. An expression mutation operator changes one expression of a target program such as an arithmetic operator or a method call [25]. I removed mutants whose faults were not detected by any testing technique in 1000 seconds (i.e., likely equivalent mutants). I also removed mutants whose faults were detected by all testing techniques in less than one second.

As real fault objects, we used the five faulty Java applications used for the experiments with CAPP [47]. Note that these programs are all real fault objects that are used in [47] and at the same time publicly available for controlled experiments [1]. `Groovy` is a concurrent data structure from a dynamic language system Groovy. `Lang` is from a math library in the Apache Commons project. `Pool-107`, `Pool-120`, and `Pool-146` are from an object-pooling library Apache Pool. `Pool-107` is the bug reported at Issue 107 in the Apache Pool bug repository. `Pool-120` and `Pool-146` are the bugs reported at Issues 120 and 146, respectively. `Groovy` and `Pool-146` have deadlock faults, and the other three objects show assertion violations as failures[12] Each of the real fault objects from SIR [1] consists of a faulty version, a test case that can replay a failure with certain thread schedules, and the patched version. As in the CAPP experiments [47], we used the faulty version of a real-fault object as the modified version, and the patched version as the original version.

Table 5.2 presents the target programs that are used for the test generation experiments. I used 65 faulty versions of 3 programs that are generated by mutations, and 5 real-world faulty programs that are used by the related work including CAPP. The third column in Table 5.2 reports the number of faulty versions used for the experiments. For the mutation objects, the numbers in the parenthesis are the number of (likely) equivalent mutants and the number of mutants killed in less than one second by all techniques. These two kinds of mutants are generated, but not used for the experiments. The fourth and the fifth columns in Table 5.2 show the number of code lines and the number of changed lines of the target program. For the mutation objects, the fourth and the fifth columns represent the number of code lines in the original program, and the average number of code lines that are modified or added in the faulty versions of each target program. The number of changed line is less than 1.5 on average because the mutation operators change one statement in most cases. [13] For the real fault objects, the fourth and

[10] Each program is used with the synchronization wrapper to provide thread-safe behaviors. For example, `ArrayList` is always wrapped with `Collections.SynchronizedList` in the test cases used for our study. I fixed all known faults of the programs before mutation.

[11] I used the following synchronization mutation operators: exchange synchronized block parameters, remove synchronized block, shrink synchronized block, and split synchronized block. I used the following expression mutation operators: access flag change, argument order change, arithmetic operator change, logical connector change, and relational operator change.

[12] Full details and the code are available at the SIR benchmark. [1]

[13] The only exception is the "exchange synchronized block parameters" mutation operator which targets a pair of syn-

the fifth columns show the number of code lines in the modified version, and the number of code lines that are modified or added in the modified version (i.e., faulty version). The last column presents the number of threads in a test execution for each program.

5.4.3 Test generation techniques

I applied Recurve and six types of testing techniques to the target programs. Recurve and CAPP [47] are specific to detecting regression faults, thus these techniques utilize the information on the original versions. In contrast, the other techniques generate test executions for the modified version without any information of the original version.

- *Random noise injection techniques*

  A random noise injection technique (calling it "RN") generates arbitrary time-delays to perturb thread scheduling of target program executions [27, 51]. A RN technique instruments a target program to insert a noise probe before every synchronization and data access statement. A probe injects a random time delay in 0 to m milliseconds with probability p whenever the probe is reached in a test execution. I used 5 milliseconds, 10 milliseconds, 15 milliseconds for m and 10%, 20%, and 30% for p to construct 9 RN techniques for the experiments.

- *Randomized scheduling technique*

  I used a random thread scheduling technique (calling it "RS"), similar to Sen [87]. RS first suspends a running thread before every synchronization and data access operation. Once all running threads are suspended, RS randomly selects and resumes one suspended thread.

- *Race-driven scheduling technique*

  I used RaceFuzzer [88] (calling it "DR") for the experiment. RaceFuzzer first predicts data race bugs in the target program, each of which is a pair of statements that may read and write (or write and write) the same variable concurrently without any synchronization. For each predicted data race bug, RaceFuzzer executes the target program while controlling a thread scheduler to make the two statements access the same variable concurrently. In this experiment, we configured RaceFuzzer to repeatedly generate test executions for a given amount of time. Note that a regression testing technique SimRT [121] uses RaceFuzzer off-then-self for generating thread schedules in test executions.[14]

- *Coverage-guided thread scheduling technique*

  I used a thread scheduling technique that achieves high concurrency coverage fast (calling it "CT") [41]. CT first estimates test requirement feasible to cover in a target program testing, and then repeats target program executions while controlling thread schedules to increase coverage achievements. CT utilizes PSet [119] together with Sync-Pair [41] following the suggestion for the coverage-driven test generation (see Chapter 3).

- *CAPP with ReEx* [47]

  As regression testing techniques, we used CAPP with a systematic testing technique ReEx[15] ReEx is

chronized blocks.

[14] I also had used AtomFuzzer and DeadlockFuzzer for the experiments, but these techniques fail to detect any faults in most cases.

[15] I used the ReEx tool at `http://mir.cs.illinois.edu/reex` but found that ReEx crashes with the mutation objects. I had reported this issue to the authors, and we fixed the bugs by ourselves for the experiments. I could not use CAPP on JPF since the tool is not publicly available.

a stateless model checker that systematically explore all thread schedules for a target program. In the experiments, we used 8 CAPP modes by combining 4 basic heuristics of impacted code identifications (`CLASS`, `METHOD`, `LINE`, `FIELD`) and 2 prioritization heuristics `ALL` and `SOME` (see Section 5.2.2). In addition, we used two ReEx search orders `Default` and `Random` which results in 16 different techniques of CAPP-ReEx. In the experiments, we provided the information of the program changes for each mutant and real-fault object as described in [47].

- *Recurve*

  I implemented Recurve for Java (15 Java classes, 10 KLOC). Recurve uses the `diff` utility to determine code changes between two versions of a target program. To measure coverage and control thread schedules at runtime, Recurve inserts scheduling probes before/after every synchronization or data access statement in the target program by using Soot [103]. Recurve is compiled and executed on Java 7 (1.7.0). Recurve determines that coverage increase is not significant if the number of newly covered test requirements at the last test execution is less than 1% of the total number of test requirements covered so far. Recurve changes the class of test targets when the coverage increase is not significant for last ten executions in a row.

  In the experiment, Recurve receives the source code of the both original and the modified versions, and the testing result of the original version which is the set of test requirements covered in 1000 seconds of test executions. I created the testing result of the original version by running the coverage guided test generation for the original version, as similar to CT.

5.4.4 Test runs

I applied each test generation technique to automatically generate test executions for 1000 seconds with each mutant or a real fault object. A testing run is generated for 1000 seconds because, according to our preliminary experiment, the results do not change much after 1000 seconds. I generate 30 testing runs for each pair of a testing technique and a target program (i.e., a mutant or a real-fault object) to obtain statistical reliability in the resulting data [82]. All experiments were performed on 30 machines with Intel i5 3.6GHz CPU and 8GB main memory, and running 64-bit Debian Linux 3.2.0.

For those techniques with dynamic analyses prior to test generation (i.e., DR, CT, and Recurve), we ran the dynamic analyses with ten executions prior to test generation, and then fed the analysis results to test generations.

5.4.5 Measurement

To assess fault detection effectiveness and efficiency, we first compute average fault detection over test generation time. The fault detection ability $F_x(p,t)$ of mutation object m and testing technique x at time t in one testing run is defined as follows:

$$F_x(p,t) = \frac{\Sigma_{m \in \mathcal{M}(p)} f_x(m,t)}{|\mathcal{M}(p)|}$$

where $\mathcal{M}(p)$ is a set of mutated faulty versions of p, $f_x(m,t) \in \{0,1\}$ indicates if the fault of a mutated faulty version m has been detected by the time instant t (1 if detected, 0 otherwise) in a testing run. If a technique crashes (e.g., due to exceeding memory) at t before detecting the fault, the fault detection is 0 for test generation time. For example, suppose that the average fault detection ability $F_{Recurve}(\texttt{ArrayList}, 100)$ is 0.94 (=17/18) where $\mathcal{M}(\texttt{ArrayList})$=$\{m_1,...,m_{18}\}$, $f_{Recurve}(m_1, 100) =$

Table 5.3: Fault detection effectiveness

Program	RN						RS	CT	DR	CAPP (default)				CAPP (random)				Recurve
	5ms		10 ms		15 ms					Any		All		Any		All		
	High	Low	High	Low	High	Low				High	Low	High	Low	High	Low	High	Low	
ArrayList	0.77	0.63	0.85	0.64	0.95	0.63	0.55	0.95	0.93	0.00	0.00	0.11	0.11	0.20	0.19	0.38	0.25	1.00
HashMap	0.81	0.55	0.92	0.54	0.91	0.49	0.29	0.92	0.58	0.15	0.15	0.16	0.15	0.32	0.32	0.33	0.32	0.97
TreeSet	0.82	0.74	0.87	0.79	0.82	0.73	0.62	0.66	0.25	0.29	0.27	0.07	0.06	0.57	0.57	0.67	0.64	0.94
Groovy	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	0.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
Lang	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
Pool-107	1.00	1.00	1.00	1.00	1.00	1.00	1.00	0.83	0.07	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
Pool-120	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
Pool-146	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	0.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00

1,..., $f_{Recurve}(m_{17}, 100) = 1$, and $f_{Recurve}\ (m_{18}, 100) = 0$ for m_1 to m_{18} of `ArrayList`. I measure the average fault detection ability of a testing technique over the 30 testing runs on each mutation object. For a real-fault object p, we measure the average fault detection ability by using $f_x(p, t)$ over 30 testing runs. I define the fault detection effectiveness of a technique for a target program as the fault detection ability at 1000 seconds.

To assess fault detection efficiency, we measure average test generation time required for each technique to achieve fault detection abilities 0.7, 0.8, 0.9, 1.0 with each target program. If no testing technique achieves fault detection ability 1.0 for an object in the study, we measured average time for achieving 70%, 80%, 90%, and 100% of the highest fault detection ability observed in the study.

5.5 Experiment results

5.5.1 RQ1. fault detection effectiveness

Table 5.3 show the fault detection results of the techniques and the studied objects. The second to the seventh column show the results of the nine random noise injection-based techniques (RN). The nine techniques are grouped according to the maximum time per noise noise (5 milliseconds, 10 milliseconds, and 15 milliseconds), and present only the highest and the lowest fault detections of each group. The eighth column is for the result of the randomized scheduling technique (RS), the ninth column for the coverage based thread scheduling technique (CT), and the tenth column for the data race-based technique (DR). The eleventh to the eighteenth columns present the results of the CAPP techniques, grouped by the search mode ('Default' or 'Random') and the context-switching condition (`ALL` or `SOME`). For each group of the CAPP technique, the table shows the highest fault detection (High) and the lowest fault detection (Low). The last column shows the result of our technique Recurve.

In overall, Table 5.3 shows that Recurve achieves higher or equivalent fault detections compared to the other techniques in the experiments. For three mutation objects (the second to the forth rows at Table 5.3), Recurve achieves much higher fault detections than all the other techniques. Recurve achieves fault detections 1.05 times (=(1.0-0.95)/0.95) to 14.7 times (=(0.94-0.06)/0.06) higher than the other techniques. The CAPP techniques achieve fault detections no more than 0.70 for all mutation objects.

For the five real-fault objects (the fifth to the ninth rows at Table 5.3), most techniques achieve the maximum fault detection (i.e., 1.00) except CT on Pool-107 and DR on Groovy, Pool-107, and Pool-146. I conjecture that these two techniques fail to detect the faults because the given test generation targets

Table 5.4: Time for achieving high levels of fault detection effectiveness (in second)

Program	70%					80%					90%					100%				
	RN	CT	DR	CAPP	Recurve	RN	CT	DR	CAPP	Recurve	RN	CT	DR	CAPP	Recurve	RN	CT	DR	CAPP	Recurve
ArrayList	36	5	8	-	9	164	7	20	-	12	595	13	80	-	24	-	-	-	-	517
HashMap	89	5	-	-	7	163	13	-	-	22	348	36	-	-	84	-	-	-	-	758
TreeSet	26	51	-	-	33	68	-	-	-	56	654	-	-	-	181	-	-	-	-	897
Groovy	12	11	-	1	11	12	11	-	1	11	12	11	-	1	12	18	12	-	1	13
Lang	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	2
Pool-107	17	595	-	1	117	30	797	-	1	169	53	-	-	1	278	80	-	-	2	704
Pool-120	1	6	1	1	3	1	6	1	1	3	1	6	1	1	4	2	8	1	1	5
Pool-146	11	11	-	1	12	11	11	-	1	13	12	12	-	1	14	13	13	-	1	16

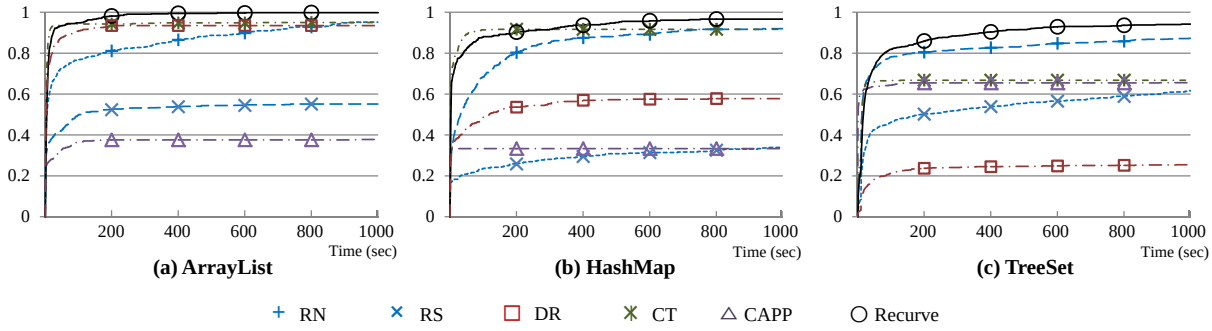


Figure 5.4: Fault detection over time per technique, for the three mutation objects

(e.g., a test requirement or a data race pattern) are not effective for generating the failure-inducing thread schedules. For example, RaceFuzzer does not generate any execution for Groovy and achieves poor fault detection (e.g., 0.00 for Groovy) because RaceFuzzer does not predict any data race in Groovy that has a deadlock bug.

Note that all CAPP techniques achieve the highest fault detections for all real-fault programs in contrast to the results with the mutation objects. I conjecture that the performance of CAPP may depend on the number of threads in a test execution, because all real-fault programs have 3 threads, while the mutation objects have 22 to 27 threads.

RaceFuzzer shows low to moderate fault detections for two mutation objects (HashMap and TreeSet) and does not achieve the maximum fault detection for two real-fault objects (Groovy and Pool-107). This result implies that the regression testing technique with the data race-based thread scheduling such as SimRT [121] may fail to detect regression faults if the regression fault is not related to data race.

5.5.2 RQ2. fault detection efficiency

Table 5.4 describes the time required for each technique to reach four levels of fault detections: 70%, 80%, 90%, and 100% of the highest fault detection observed in the study. For example, as the highest fault detection of TreeSet is 0.94 in the experiment (see the last column of fourth row of Table 5.3), a number of the second to the sixth columns is the first time for a technique to reach fault detection 0.658 (=0.7×0.94). The columns of RN represent the shortest times among the nine random noise injection-based techniques for each level. Similarly, the columns of CT, DR, CAPP, and Recurve represent their shortest times at each fault detection level. I omitted RS since the technique does not achieve fault detection of 70% for all three mutation objects.

For the three mutation objects, a number at $x\%$ fault detection level in Table 5.4 indicates the time for a technique to generate a failing execution for $x\%$ of the mutants. For reaching the 100% level, only Recurve achieves the fault detection level in 1000 seconds. For the lower levels, Recurve takes less time to achieve all cases than the RN techniques except `TreeSet` for 70% level. Compared to the CT technique, Recurve spent more times for `ArrayList` and `HashMap` for 70% to 90% levels but CT failed to achieve fault detection level higher than 70% for `TreeSet`.

Figure 5.4 is the plots of fault detection over times per technique for the three mutation objects. [16] The plots in the figures show that Recurve and RN continuously increase fault detection over 1000 seconds. Conversely, the other techniques (i.e., RS, DR, CT, and CAPP) do not effectively increase fault detection once the technique reaches certain levels of fault detections (i.e., saturation points). For example, CT achieves the 90% level for HashMap in 36 seconds, and no further increase is observed for the remaining time. This result implies that the fault detection capability of RS, DR, CT and CAPP can vary depending on target concurrency bugs. Thus, we conjecture that Recurve and RN can be more suitable for broad range of concurrency bugs than the other techniques.

For the real-fault objects, a number n at $x\%$ fault detection level in Table 5.4 means that the probability for a technique to generate the failing execution in n seconds is $x\%$. Note that all techniques except for CAPP take at least 10 seconds to detect the deadlock faults in Groovy and Pool-146 (because timeout to determine deadlock is 10 seconds). CAPP can immediately detect a deadlock by checking state condition.

The results shows that the CAPP techniques achieve the maximum fault detections (i.e., 1.0) for all five real-fault objects in 2 seconds. For Groovy and Pool-146, RN, CT, Recurve takes no more than 20 seconds to detect the deadlock faults. For Lang and Pool-120, RN, CT, DR, and Recurve detect the faults in no more than 8 seconds. For Pool-107, Recurve requires far more time to detect the failure than RN and CAPP. I found that the fault of Pool-107 induces a failure only if a context-switch occurs at a specific iteration of a loop in a thread. CT does not reach the 90% level in 1000 seconds, and CT takes more time for reaching 70% and 80% than Recurve. I conjecture that the limitation of coverage-based thread scheduling is more severe for CT because the PSet and Sync-Pair metric used by CT is more coarse-grained than the CC metric used by Recurve.

5.5.3 Impact of test target prioritization

To check the benefit of the test target prioritization scheme, we additionally experiment Recurve$_{NP}$, a variant of Recurve that does not prioritize the test targets, but uses all test targets (i.e., Class A $\cup$ Class B $\cup$ Class C) from the beginning. The experiment results show that Recurve$_{NP}$ shows lower fault detection effectiveness than Recurve for HashMap (as 0.95) and TreeSet (as 0.92). For all three mutation objects, Recurve$_{NP}$ is slower to reach high levels of fault detections. For example, with TreeSet, Recurve$_{NP}$ takes 33, 77, 231 seconds for reaching 70%, 80%, 90% levels, respectively and does not reach 100% level within 1000 seconds. This result implies that the test target prioritization contributes the high fault detection efficiency of Recurve substantially.

[16] RN in each figure is the one with the highest fault detection among the nine random noise-based techniques. Similarly, we plot the CAPP technique that achieves the highest fault detections for each object.

5.6 Summary of this chapter

This chapter presents a concurrency coverage-based thread schedule generation technique for regression testing of multithreaded programs. I propose a new coverage metric effective for regression testing of multithreaded programs, and present a test generation technique that detect and target the changed behavior of a target program based on the new coverage metric. The experiment result shows that the proposed test generation technique is more effective and efficient for detecting regression faults than existing regression testing techniques and conventional testing techniques for multithreaded programs.

Chapter 6. Conclusion

Despite the widespread of multithreaded programming in practice, effective testing of multithreaded programs is still challenging for developers since no proper testing methodologies or tools have been supported. In this dissertation, I show that generating multithreaded program tests to achieve high concurrency coverage is effective and efficient at detecting concurrency errors in real-world multithreaded programs. As a first step toward supporting effective multithreaded program testing, I show empirical evidence that the existing concurrency coverage metrics are actually useful to guide test generation to improve fault detection ability. This result implies that the existing concurrency coverage metrics are proper testing effectiveness predictors and also proper test targets for automated testing techniques. This empirical investigation is the first comprehensive study on the testing effectiveness of concurrency coverage metrics.

Based on the positive empirical evidence on concurrency coverage metrics, I present new automated testing techniques that utilize concurrency coverage metrics to generate highly effective and efficient multithreaded program tests. First, in order to detect concurrency errors effectively and efficiently, I present CUVE, a new testing technique that achieves high concurrency coverage quickly. In particular, I propose a new concurrency coverage metric for test generation called combinatorial concurrency coverage, and new thread scheduling algorithms that dynamically coordinate execution order over threads to make test executions that can achieve high combinatorial coverage efficiently. Through experiments with various buggy multithreaded programs, I show that CUVE detects concurrent errors in a more effective and efficient manner than do the existing multithreaded program testing techniques. In addition, I extend the concurrency coverage-based testing techniques to support effective regression testing of evolving multithreaded programs. In particular, I develop Recurve, a regression testing technique for multithreaded programs, that utilizes a concurrency coverage metric to effectively target the new program behaviors introduced by code changes. The empirical evaluation results show that the proposed technique is more effective and efficient at detecting regression faults in multithreaded programs than are existing techniques.

Concurrency coverage metrics are promising testing methods that can associate advanced program analyses and testing practices because they provide intuitive and concrete abstractions of a target program behavior that can be understood by both developers and automated analysis techniques. I expect that many sophisticated coverage-based testing ideas originally developed in the sequential program domain can be extended to the multithreaded program domain. concurrency coverage metrics are promising methods to associate advanced program As future work, I will investigate effective input value generation for multithreaded program testing. Together with thread schedules, program input value is an important factor that influences multithreaded program testing effectiveness. I plan to combine coverage-based thread schedule generation and symbolic execution for effective and efficient test input value generation.

References

[1] Software-artifact Infastructure Repository. `http://sir.unl.edu`.

[2] S. Adve. Data races are evil with no exceptions: Technical perspective. *Communications of the ACM*, 53(11):84–84, 2010.

[3] J. H. Andrews, L. C. Briand, and Y. Labiche. Is mutation an appropriate tool for testing experiments? In *Proceedings of the International Conference on Software Engineering (ICSE)*, 2005.

[4] J. H. Andrews, L. C. Briand, Y. Labiche, and A. S. Namin. Using mutation analysis for assessing and comparing testing coverage criteria. *IEEE Transactions on Software Engineering (TSE)*, 32(8):608–624, Aug. 2006.

[5] C. Artho, K. Havelund, and A. Biere. High-level data races. In *Proceedings of the International Workshop on Verification and Validation of Enterprise Information Systems (VVIES)*, 2003.

[6] C. Artho, K. Havelund, and A. Biere. Using block-local atomicity to detect stale-value concurrency errors. In *Proceedings of the International Symposium on Automated Technology for Verification and Analysis (ATVA)*, 2004.

[7] E. Bodden and K. Havelund. Aspect-oriented race detection in Java. *IEEE Transactions on Software Engineering (TSE)*, 36(4):509–527, Jul 2010.

[8] H.-J. Boehm. Position paper: nondeterminism is unavoidable, but data races are pure evil. In *Proceedings of ACM Workshop on Relaxing Synchronization for Multicore and Manycore Scalability (RACES)*, 2012.

[9] J. S. Bradbury, J. R. Cordy, and J. Dingel. Mutation operators for concurrency Java (J2SE 5.0). In *Proceedings of the Workshop on Mutation Analysis (MUTATION)*, 2006.

[10] J. S. Bradbury and K. Jalbert. Defining a catalog of programming and anti-patterns for concurrent Java. In *Proceedings of the International Workshop on Software Patterns and Quality (SPAQu)*, 2009.

[11] A. Bron, E. Farchi, Y. Magid, Y. Nir, and S. Ur. Applications of synchronization coverage. In *Proceedings of the ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, 2005.

[12] S. Burckhardt, P. Kothari, M. Musuvathi, and S. Nagarakatte. A randomized scheduler with probabilistic guarantees of finding bugs. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, Mar. 2010.

[13] J. Burnim, K. Sen, and C. Stergiou. Testing concurrent programs on relaxed memory models. In *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA)*, 2011.

[14] C. Cadar, D. Dunbar, and D. Engler. KLEE: unassisted and automatic generation of high-coverage tests for complex systems programs. In *Proceedings of the USENIX Conference on Operating Systems Design and Implementation (OSDI)*, 2008.

[15] X. Cai and M. R. Lyu. The effect of code coverage on fault detection under different testing profiles. In *Proceedings of International Workshop of Advances in Model-based Testing (A-MOST)*, 2005.

[16] F. Chen and G. Rosu. Parametric and sliced causality. In *Proceedings of the International Conference on Computer Aided Verification (CAV)*, 2007.

[17] Q. Chen, L. Wang, Z. Yang, and S. D. Stoller. HAVE: integrated dynamic and static analysis for atomicity violations. In *Proceedings of the International Conference on Fundamental Approaches to Software Engineering (FASE)*, 2009.

[18] T.-H. Chen, M. Nagappan, E. Shihab, and A. E. Hassan. An empirical study of dormant bugs. In *Proceedings of the Working Conference on Mining Software Repositories (MSR)*, 2014.

[19] L. Chew and D. Lie. Kivati: fast detection and prevention of atomicity violations. In *Proceedings of the European Conference on Computer Systems (EuroSys)*, 2010.

[20] J. D. Choi, K. Lee, A. Loginov, R. O'Callahan, V. Sarkar, and M. Sridharan. Efficient and precise datarace detection for multithreaded object-oriented programs. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2002.

[21] M. Christiaens and K. D. Bosschere. TRaDe, a topological approach to on-the-fly race detection in Java programs. In *Proceedings of the Symposium on JavaTM Virtual Machine Research and Technology Symposium (JVM)*, 2001.

[22] D. Deng, W. Zhang, and S. Lu. Efficient concurrency-bug detection across inputs. *ACM SIGPLAN Notices*, 48(10):785–802, 2013.

[23] D. Deng, W. Zhang, B. Wang, P. Zhao, and S. Lu. Understanding the interleaving-space overlap across inputs and software versions. In *Proceedings of the USENIX Workshop on Hot Topics in Parallelism (HotPar)*, 2012.

[24] R. J. Dias, V. Pessanha, and J. M. Lourenço. Precise detection of atomicity violations. In *Proceedings of the International Conference on Hardware and Software: Verification and Testing (HVC)*, 2012.

[25] H. Do and G. Rothermel. A controlled experiment assessing test case prioritization techniques via mutation faults. In *Proceedings of the IEEE International Conference on Software Maintenance (ICSM)*, 2005.

[26] M. B. Dwyer, S. Person, and S. G. Elbaum. Controlling factors in evaluating path-sensitive error detection techniques. In *Proceedings of the ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE)*, 2006.

[27] O. Edelstein, E. Farchi, Y. Nir, G. Ratsaby, and S. Ur. Multithreaded Java program test generation. In *Proceedings of the Joint ACM-ISCOPE Conference on Java Grande (JGI)*, 2001.

[28] M. Emmi, S. Qadeer, and Z. Rakamarić. Delay-bounded scheduling. In *Proceedings of the 38th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages (POPL)*, 2011.

[29] D. Engler and K. Ashcraft. RacerX: effective, static detection of race conditions and deadlocks. In *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP)*, 2003.

[30] J. Fiedor, B. Křena, Z. Letko, and T. Vojnar. A uniform classification of common concurrency errors. In *Proceedings of the International Conference on Computer Aided Systems Theory (EUROCAST)*, 2012.

[31] C. Flanagan and S. N. Freund. Type-based race detection for java. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2000.

[32] C. Flanagan and S. N. Freund. Atomizer: a dynamic atomicity checker for multithreaded programs. *Science of Computer Programming*, 71(2):89–109, 2008.

[33] C. Flanagan and S. N. Freund. FastTrack: efficient and precise dynamic race detection. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2009.

[34] C. Flanagan and S. N. Freund. The RoadRunner dynamic analysis framework for concurrent programs. In *Proceedings of Workshop on Program Analysis for Software Tools and Engineering (PASTE)*, 2010.

[35] C. Flanagan, S. N. Freund, and J. Yi. Velodrome: a sound and complete dynamic atomicity checker for multithreaded programs. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2008.

[36] M. Gligoric, V. Jagannath, and D. Marinov. MutMut: efficient exploration for mutation testing of multithreaded code. In *Proceedings of International Conference on Software Testing, Verification and Validation (ICST)*, 2010.

[37] P. Godefroid, N. Klarlund, and K. Sen. DART: directed automated random testing. *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2005.

[38] P. Godefroid and N. Nagappan. Concurrency at Microsoft: an exploratory survey. In *Proceedings of CAV Workshop on Exploiting Concurrency Efficiently and Correctly*, 2008.

[39] C. Hammer, J. Dolby, M. Vaziri, and F. Tip. Dynamic detection of atomic-set-serializability violations. In *Proceedings of the International Conference on Software Engineering (ICSE)*, 2008.

[40] D. P. Helmbold and C. E. McDowell. A taxonomy of race conditions. *Journal of Parallel and Distributed Computing*, 33(2):159–164, 1996.

[41] S. Hong, J. Ahn, S. Park, M. Kim, and M. J. Harrold. Testing concurrent program to achieve high synchronization coverage. In *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA)*, 2012.

[42] S. Hong and M. Kim. A survey of race bug detection techniques for multithreaded programmes. *Software Testing, Verification and Reliability (STVR)*, 25(3):191–217, 2015.

[43] S. Hong, M. Staats, J. Ahn, M. Kim, and G. Rothermel. The impact of concurrent coverage metrics on testing effectiveness. In *Proceedings of the IEEE International Conference on Software Testing, Verification and Validation (ICST)*, 2013.

[44] S. Hong, M. Staats, J. Ahn, M. Kim, and G. Rothermel. Are concurrency coverage metrics effective for testing: A comprehensive empirical investigation. *Software Testing, Verification and Reliability (STVR)*, 25(4):334–370, 2015.

[45] V. Hrubá, B. Křena, Z. Lekto, S. Ur, and T. Vojnar. Testing of concurrent programs using genetic algorithms. In *Proceedings of the International Conference on Search Based Software Engineering (SSBSE)*, 2012.

[46] J. Huang and C. Zhang. Persuasive prediction of concurrency access anomalies. In *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA)*, 2011.

[47] V. Jagannath, Q. Luo, and D. Marinov. Change-aware preemption prioritization. In *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA)*, 2011.

[48] P. Joshi, M. Naik, C.-S. Park, and K. Sen. CalFuzzer: an extensible active testing framework for concurrent programs. In *Proceedings of International Conference on Computer Aided Verification (CAV)*, 2009.

[49] P. Joshi, C.-S. Park, K. Sen, and M. Naik. A randomized dynamic program analysis technique for detecting real deadlocks. *ACM SIGPLAN Notices*, 44(6):110–120, 2009.

[50] P. Kvam and B. Vidakovic. *Nonparametric Statistics with Applications to Science and Engineering.* Wiley, 2007.

[51] B. Křena, Z. Letko, T. Vojnar, and S. Ur. A platform for search-based testing of concurrent software. In *Proceedings of the Workshop on Parallel and Distributed Systems: Testing, Analysis, and Debugging (PADTAD)*, 2010.

[52] Z. Lai, S.-C. Cheung, and W. K. Chan. Detecting atomic-set serializability violations in multithreaded programs through active randomized testing. In *Proceedings of the ACM/IEEE International Conference on Software Engineering (ICSE)*, 2010.

[53] Z. Letko, T. Vojnar, and B. Křena. AtomRace: data race and atomicity violation detector and healer. In *Proceedings of the Workshop on Parallel and Distributed Systems: Testing, Analysis, and Debugging (PADTAD)*, 2008.

[54] D. Li, W. Srisa-an, and M. B. Dwyer. SOS: saving time in dynamic race detection with stationary analysis. In *Proceedings of the ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications (OOPSLA)*, 2011.

[55] B. Long and P. Strooper. A classification of concurrency failures in Java components. In *Proceedings of the International Symposium on Parallel and Distributed Processing (IPDPS)*, 2003.

[56] S. Lu, W. Jiang, and Y. Zhou. A study of interleaving coverage criteria. In *Proceedings of the Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE)*, 2007.

[57] S. Lu, S. Park, C. Hu, X. Ma, W. Jiang, Z. Li, R. A. Popa, and Y. Zhou. MUVI: automatically inferring multi-variable access correlations and detecting related semantic and concurrency bugs. In *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP)*, 2007.

[58] S. Lu, S. Park, E. Seo, and Y. Zhou. Learning from mistakes: a comprehensive study on real world concurrency bug characteristics. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2008.

[59] S. Lu, J. Tucek, F. Qing, and Y. Zhou. AVIO: Detecting atomicity violations via access interleaving invariants. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2006.

[60] B. Lucia, L. Ceze, and K. Strauss. ColorSafe: architectural support for debugging and dynamically avoiding multi-variable atomicity violations. In *Proceedings of the Annual International Symposium on Computer Architecture (ISCA)*, 2010.

[61] B. Lucia, J. Devietti, K. Strauss, and L. Ceze. Atom-Aid: Detecting and surviving atomicity violations. In *Proceedings of the Annual International Symposium on Computer Architecture (ISCA)*, 2008.

[62] C. Mallows. Some comments on *Cp*. *Technometrics*, 1973.

[63] D. Marino, M. Musuvathi, and S. Narayanasamy. LiteRace: effective sampling for lightweight data-race detection. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2009.

[64] V. Mekkat, A. Holey, and A. Zhai. Accelerating data race detection utilizing on-chip data-parallel cores. In *Proceedings of the International Conference on Runtime Verification (RV)*, 2013.

[65] M. Musuvathi and S. Qadeer. Iterative context bounding for systematic testing of multithreaded programs. *SIGPLAN Not.*, 42(6), June 2007.

[66] A. Muzahid, N. Otsuki, and J. Torrellas. AtomTracker: a comprehensive approach to atomic region inference and violation detection. In *Proceedings of the Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2010.

[67] S. Nagarakatte, S. Burckhardt, M. M. K. Martin, and M. Musuvathi. Multicore acceleration of priority-based schedulers for concurrency bug detection. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2012.

[68] M. Naik, A. Aiken, and J. Whaley. Effective static race detection for Java. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2006.

[69] A. S. Namin and J. H. Andrews. The influence of size and coverage on test suite effectiveness. In *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA)*, 2009.

[70] R. H. B. Netzer and B. P. Miller. What are race conditions?: some issues and formalizations. *ACM Letters on Programming Languages and Systems*, 1(1):74–88, Mar 1992.

[71] A. Nistor, Q. Luo, M. Pradel, T. R. Gross, and D. Marinov. BALLERINA: automatic generation and clustering of efficient random unit tests for multithreaded code. In *Proceedings of the International Conference on Software Engineering (ICSE)*, 2012.

[72] R. O'Callahan and J. D. Choi. Hybrid dynamic data race detection. In *Proceedings of the ACM Symposium on Principles and Practice of Parallel Programming (PPoPP)*, 2003.

[73] S. Okur and D. Dig. How do developers use parallel libraries? In *Proceedings of the ACM SIGSOFT Foundation of Software Engineering (FSE)*, 2012.

[74] C. Pacheco, S. K. Lahiri, M. D. Ernst, and T. Ball. Feedback-directed random test generation. In *Proceedings of the International Conference on Software Engineering (ICSE)*, 2007.

[75] C. S. Park and K. Sen. Randomized active atomicity violation detection in concurrent programs. In *Proceedings of the ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE)*, 2008.

[76] S. Park, S. Lu, and Y. Zhou. CTrigger: exposing atomicity violation bugs from their hiding places. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2009.

[77] S. Park, R. W. Vuduc, and M. J. Harrold. Falcon: fault localization in concurrent programs. In *Proceedings of the International Conference on Software Engineering (ICES)*, 2010.

[78] P. Piwowarski, M. Ohba, and J. Caruso. Coverage measurement experience during function test. In *Proceedings of the International Conference on Software Engineering (ICSE)*, 1993.

[79] E. Pozniansky and A. Schuster. MultiRace: efficient on-the-fly data race detection in multithreaded C++ programs. *Concurrency and Computation: Practice and Experience*, 19(3):327–340, 2007.

[80] S. Qadeer and S. Tasiran. Runtime verification of concurrency-specific correctness. *Software Tools for Technology Transfer*, 14(3):291–305, 2012.

[81] A. Raza. A review of race detection mechanisms. In *Proceedings of the International Computer Science Conference on Theory and Applications (CSR)*, 2006.

[82] J. A. Rice. *Mathematical Statistics and Data Analysis*. Cengage Learning, 2006.

[83] C. Sadowski, J. Yi, and S. Kim. The evolution of data races. In *Proceedings of the IEEE Working Conference on Mining Software Repositories (MSR)*, 2012.

[84] S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, and T. Anderson. Eraser: A dynamic data race detector for multithreaded programs. *ACM Transactions on Computer Systems (TOCS)*, 15(4):391–411, 1997.

[85] J. Schimmel, K. Molitorisz, and W. F. Tichy. An evaluation of data race detectors using bug repositories. In *Proceedings of the Federated Conference on Computer Science and Information Systems*, 2013.

[86] P. Seibel. *Coders at work*. Apress, 2009.

[87] K. Sen. Effective random testing of concurrent programs. In *Proceedings of the IEEE International Conference on Automated Software Engineering (ASE)*, 2007.

[88] K. Sen. Race directed random testing of concurrent programs. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2008.

[89] K. Sen and G. Agha. A race-detection and flipping algorithm for automated testing of multithreaded programs. In *Proceedings of the International Conference on Hardware and Software: Verification and Testing (HVC)*, 2006.

[90] K. Serebryany and T. Iskhodzhanov. ThreadSanitizer – data race detection in practice. In *Proceedings of the Workshop on Binary Instrumentation and Applications (WBIA)*, 2009.

[91] T. Sheng, N. Vachharajani, S. Eranian, R. Hundt, W. Chen, and W. Zheng. RACEZ: a lightweight and non-invasive race detection tool for production applications. In *Proceedings of the International Conference on Software Engineering (ICSE)*, 2011.

[92] E. Sherman, M. B. Dwyer, and S. Elbaum. Saturation-based testing of concurrent programs. In *Proceedings of the Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering(ESEC/FSE)*, 2009.

[93] Y. Shi, S. Park, Z. Yin, S. Lu, Y. Zhou, W. Chen, and W. Zheng. Do I use the wrong definition? DeFuse: definition-use invariants for detecting concurrency and sequential bugs. In *Proceedings of the ACM International Conference on Object-oriented Programming Systems Languages and Applications (OOPSLA)*, 2010.

[94] F. Sorrentino, A. Farzan, and P. Madhusudan. PENELOPE: weaving threads to expose atomicity violations. In *Proceedings of the ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE)*, 2010.

[95] S. R. S. Souza, M. A. S. Brito, R. A. Silva, P. S. L. Souza, and E. Zaluska. Research in concurrent software testing: a systematic review. In *Proceedings of the Workshop on Parallel and Distributed Systems: Testing, Analysis, and Debugging (PADTAD)*, 2011.

[96] S. D. Stoller. Testing concurrent Java programs using randomized scheduling. In *Proceedings of the International Workshop on Runtime Verification (RV)*, 2002.

[97] W. N. Sumner, C. Hammer, and J. Dolby. Marathon: Detecting atomic-set serializability violations with conflict graphs. In *Proceedings of the International Conference on Runtime Verification (RV)*, 2011.

[98] S. Tasiran, M. E. Keremoglu, and K. Muslu. Location pairs: a test coverage metric for shared-memory concurrent programs. *Empirical Software Engineering (ESE)*, 17(3):129–165, 2012.

[99] G. M. Tchamgoue, O.-K. Ha, K.-H. Kim, and Y.-K. Jun. A taxonomy of concurrency bugs in event-driven programs. In *Proceedings of International Conference on Advanced Software Engineering and Its Applications (ASEA)*, 2011.

[100] V. Terragni, S.-C. Cheung, and C. Zhang. Recontest: Effective regression testing of concurrent programs. In *Proceedings of the International Conference on Software Engineering (ICSE)*, 2015.

[101] N. Tracey, J. Clark, K. Mander, and J. McDermid. An automated framework for structural test-data generation. In *Proceedings of the IEEE International Conference on Automated Software Engineering (ASE)*, 1998.

[102] E. Trainin, Y. Nir-Buchbinder, R. Tzoref-Brill, A. Zlotnick, S. Ur, and E. Farchi. Forcing small models of conditions on program interleaving for detection of concurrent bugs. In *Proceedings of the Workshop on Parallel and Distributed Systems: Testing, Analysis, and Debugging (PADTAD)*, 2009.

[103] R. Vallée-Rai, P. Co, E. Gagnon, L. Hendren, P. Lam, and V. Sundaresan. Soot - a Java bytecode optimization framework. In *Proceedings of the Center for Advanced Studies Conference (CASCON)*, 1999.

[104] M. Vaziri, F. Tip, and J. Dolby. Associating synchronization constraints with data in an object-oriented language. In *Proceedings of the Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, 2006.

[105] W. Visser, K. Havelund, G. Brat, S. Park, and F. Lerda. Model checking programs. *Automated Software Engineering Journal*, 10(2):203–232, 2003.

[106] C. von Praun and T. Gross. Static detection of atomicity violations in object-oriented programs. *Journal of Object Technology*, 3(2):103–122, Mar-Apr 2004.

[107] C. von Praun and T. R. Gross. Object race detection. In *Proceedings of the ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications (OOPSLA)*, 2001.

[108] J. W. Voung, R. Jhala, and S. Lerner. RELAY: static race detection on millions of lines of code. In *Proceedings of the joint meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering (ESEC-FSE)*, 2007.

[109] C. Wang, M. Said, and A. Gupta. Coverage guided systematic concurrency testing. In *Proceedings of the International Conference on Software Engineering (ICSE)*, 2011.

[110] C. Wang, Y. Yang, A. Gupta, and G. Gopalakrishnan. Dynamic model checking with property driven pruning to detect race conditions. In *Proceedings of the International Symposium on Automated Technology for Verification and Analysis (ATVA)*, 2008.

[111] L. Wang and S. D. Stoller. Accurate and efficient runtime detection of atomicity errors in concurrent programs. In *Proceedings of the ACM Symposium on Principles and Practice of Parallel Programming (PPoPP)*, 2006.

[112] L. Wang and S. D. Stoller. Runtime analysis for atomicity for multithreaded programs. *IEEE Transactions on Software Engineering (TSE)*, 32(2):99–110, 2006.

[113] X. Xie, J. Xue, and J. Zhang. ACCULOCK: accurate and efficient detection of data races. *Software–Practice and Experience*, 43(5):543–576, 2012.

[114] Z. Xu, Y. Kim, M. Kim, G. Rothermel, and M. Cohen. Directed test suite augmentation: Techniques and tradeoffs. In *Proceedings of the ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE)*, 2010.

[115] C. D. Yang, A. L. Souter, and L. L. Pollock. All-du-path coverage for parallel programs. In *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA)*, 1998.

[116] J. Yi, C. Sadowski, and C. Flanagan. Cooperative reasoning for preemptive execution. In *Proceedings of the ACM Symposium on Principles and Practice of Parallel Programming (PPoPP)*, 2011.

[117] Z. Yin, D. Yuan, Y. Zhou, S. Pasupathy, and L. Bairavasundaram. How do fixes become bugs? In *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of Software Engineering (ESEC/FSE)*, 2011.

[118] S. Yoo and M. Harman. Regression testing minimization, selection and prioritization: a survey. *Software Testing, Verification and Reliability (STVR)*, 22(2):67–120, 2012.

[119] J. Yu and S. Narayanasamy. A case for an interleaving constrained shared-memory multi-processor. In *Proceedings of the Annual International Symposium on Computer Architecture (ISCA)*, 2009.

[120] J. Yu, S. Narayanasamy, C. Pereira, and G. Pokam. Maple: a coverage-driven testing tool for multithreaded programs. In *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications (OOPSLA)*, 2012.

[121] T. Yu, W. Srisa-an, and G. Rothermel. SimRT: an automated framework to support regression testing for data races. In *Proceedings of the International Conference on Software Engineering (ICSE)*, 2014.

[122] Y. Yu, T. Rodeheffer, and W. Chen. RaceTrack: efficient detection of data race conditions via adaptive tracking. In *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP)*, 2005.

[123] R. Zeng, Z. Sun, S. Liu, and X. He. McPatom: a predictive analysis tool for atomicity violation using model checking. In *Proceedings of the International Workshop on Model Checking Software (SPIN)*, 2012.

[124] H. Zhu, P. Hall, and J. May. Software unit test coverage and adequacy. *ACM Computing Surveys (CSUR)*, 29(4):366–427, 1997.

Appendices

Chapter A. Formal Definitions of Concurrency Coverage Metrics

In this chapter I formally present the coverage satisfaction relation definitions of the eight concurrency coverage metrics (Section 2.2.2). I use the thread model presented in Section 4.3.2 with the following additional functions on an execution E:

- $\texttt{lock-holder}_E\texttt{(i,}m\texttt{)}$ returns the thread of the `lock-hold` action that holds lock m at execution of $\sigma[i]$.
- $\texttt{last-lock}_E\texttt{(i,}m\texttt{)}$ returns an index j such that $\sigma[j]$ is the $lock-hold$ action that most recently holds lock m at execution of $\sigma[i]$.
- $\texttt{last-write}_E\texttt{(i,}v\texttt{)}$ returns an index j such that $\sigma[j]$ is the *write* action that most recently writes variable v at execution of $\sigma[i]$.

Definition 10 (Blocked Test Requirement Satisfaction Condition). *For an execution $E = \langle s_0, \sigma\rangle$ and a Blocked test requirement Blocked(l), $E \models$ Blocked(l) if there exists $\sigma[i]$ such that* $\texttt{loc}(\sigma[i]) = l \wedge \texttt{operator}(\sigma[i]) = \texttt{lock-acquire} \wedge \exists\, t \in T.(t \neq \texttt{thread}(\sigma[i]) \wedge \texttt{lock-holder}_E(i,\texttt{operand}(\sigma[i])) = t)$.

Definition 11 (Blocked-Pair Test Requirement Satisfaction Condition). *For an execution $E = \langle s_0, \sigma\rangle$ and a Blocked-Pair test requirement Blocked-Pair(l_1, l_2), $E \models$ Blocked-Pair(l_1, l_2) if there exist an action $\sigma[i]$ and another $\sigma[j]$ $(i < j)$ such that* $\texttt{loc}(\sigma[i]) = l_1 \wedge \texttt{operator}(\sigma[i]) = \texttt{lock-hold} \wedge \texttt{loc}(\sigma[j]) = l_2 \wedge \texttt{operator}(\sigma[j]) = \texttt{lock-acquire} \wedge \texttt{last-lock}_E(j,\texttt{operand}(\sigma[j])) = \sigma[i]$.

Definition 12 (Blocking Test Requirement Satisfaction Condition). *For an execution $E = \langle s_0, \sigma\rangle$ and a Blocking test requirement Blocking(l), $E \models$ Blocking(l) if there exist an action $\sigma[i]$ and an action $\sigma[j]$ $(i < j)$ such that* $\texttt{loc}(\sigma[i]) = l_1 \wedge \texttt{operator}(\sigma[i]) = \texttt{lock-hold} \wedge \texttt{operator}(\sigma[j]) = \texttt{lock-acquire} \wedge \texttt{lock-holder}_E(i,\texttt{operand}(\sigma[i])) = \texttt{thread}(\sigma[i]) \wedge \texttt{last-lock}_E(j,\texttt{operand}\ (\sigma[j])) = \texttt{loc}(\sigma[i])$.

Definition 13 (Follows Test Requirement Satisfaction Condition). *For an execution $E = \langle s_0, \sigma\rangle$ and a Follows test requirement Follows(l_1, l_2), $E \models$ Follows(l_1, l_2) if there exist an action $\sigma[i]$ and an action $\sigma[j]$ $(i < j)$ such that* $\texttt{loc}(\sigma[i]) = l_1 \wedge \texttt{operator}(\sigma[i]) = \texttt{lock-hold} \wedge \texttt{loc}(\sigma[j]) = l_1 \wedge \texttt{operator}(\sigma[j]) = \texttt{lock-hold} \wedge \texttt{thread}(\sigma[i]) \neq \texttt{thread}(\sigma[j]) \wedge \texttt{last-lock}_E(j,\texttt{operand}(\sigma[j])) = \texttt{loc}(\sigma[i])$.

Definition 14 (Sync-Pair Test Requirement Satisfaction Condition). *For an execution $E = \langle s_0, \sigma\rangle$ and a Sync-Pair test requirement Sync-Pair(l_1, l_2), $E \models$ Sync-Pair(l_1, l_2) if there exist an action $\sigma[i]$ and another action $\sigma[j]$ $(i < j)$ such that* $\texttt{loc}(\sigma[i]) = l_1 \wedge \texttt{operator}(\sigma[i]) = \texttt{lock-hold} \wedge \texttt{loc}(\sigma[j]) = l_1 \wedge \texttt{operator}(\sigma[j]) = \texttt{lock-hold} \wedge \texttt{last-lock}_E(j,\texttt{operand}(\sigma[j])) = \texttt{loc}(\sigma[i])$.

Definition 15 (Def-Use Test Requirement Satisfaction Condition). *For an execution $\langle s_0, \sigma\rangle$ and a Def-Use test requirement Def-Use(l_1, l_2), $E \models$ Def-Use(l_1, l_2) if there exist an action $\sigma[i]$ and an action $\sigma[j]$ $(i < j)$ such that* $\texttt{loc}(\sigma[i]) = l_1 \wedge \texttt{operator}(\sigma[i]) = \texttt{write} \wedge \texttt{loc}(\sigma[j]) = l_2 \wedge (\texttt{operator}(\sigma[j]) = \texttt{read} \vee \texttt{operator}(\sigma[i]) = \texttt{write}) \wedge \texttt{last-write}_E(j,\texttt{operand}(\sigma[i])) = \texttt{loc}(\sigma[i])$.

Definition 16 (L-Def Test Requirement Satisfaction Condition). *For an execution $\langle s_0, \sigma\rangle$ and a L-Def test requirement L-Def(l), $E \models$ L-Def(l) if there exists an action $\sigma[i]$ and an action $\sigma[j]$ $(i < j)$ such*

that $\texttt{loc}(\sigma[j]) = l \wedge \texttt{operator}(\sigma[j]) = \texttt{read} \wedge \sigma[i] = \texttt{last-write}_E(\texttt{j},\texttt{operand}(\sigma[j])) \wedge \texttt{thread}(\sigma[i]) = \texttt{thread}(\sigma[j])$

Definition 17 (R-Def Test Requirement Satisfaction Condition)**.** *For an execution* $\langle s_0, \sigma \rangle$ *and a R-Def test requirement R-Def(l), E* $\models$ *R-Def(l) if there exists an action* $\sigma[i]$ *and an action* $\sigma[j]$ $(i < j)$ *such that* $\texttt{loc}(\sigma[j]) = l \wedge \texttt{operator}(\sigma[j]) = \texttt{write} \wedge i = \texttt{last-write}_E(\texttt{j},\texttt{operand}(\sigma[j])) \wedge \texttt{thread}(\sigma[i]) \neq \texttt{thread}(\sigma[j])$

Definition 18 (PSet Test Requirement Satisfaction Condition)**.** *For an execution* $E = \langle s_0, \sigma \rangle$ *and a PSet(*l_1*,* l_2*) test requirement,* $E \models$ *PSet(*l_1*,* l_2*) for the following the three situations:*

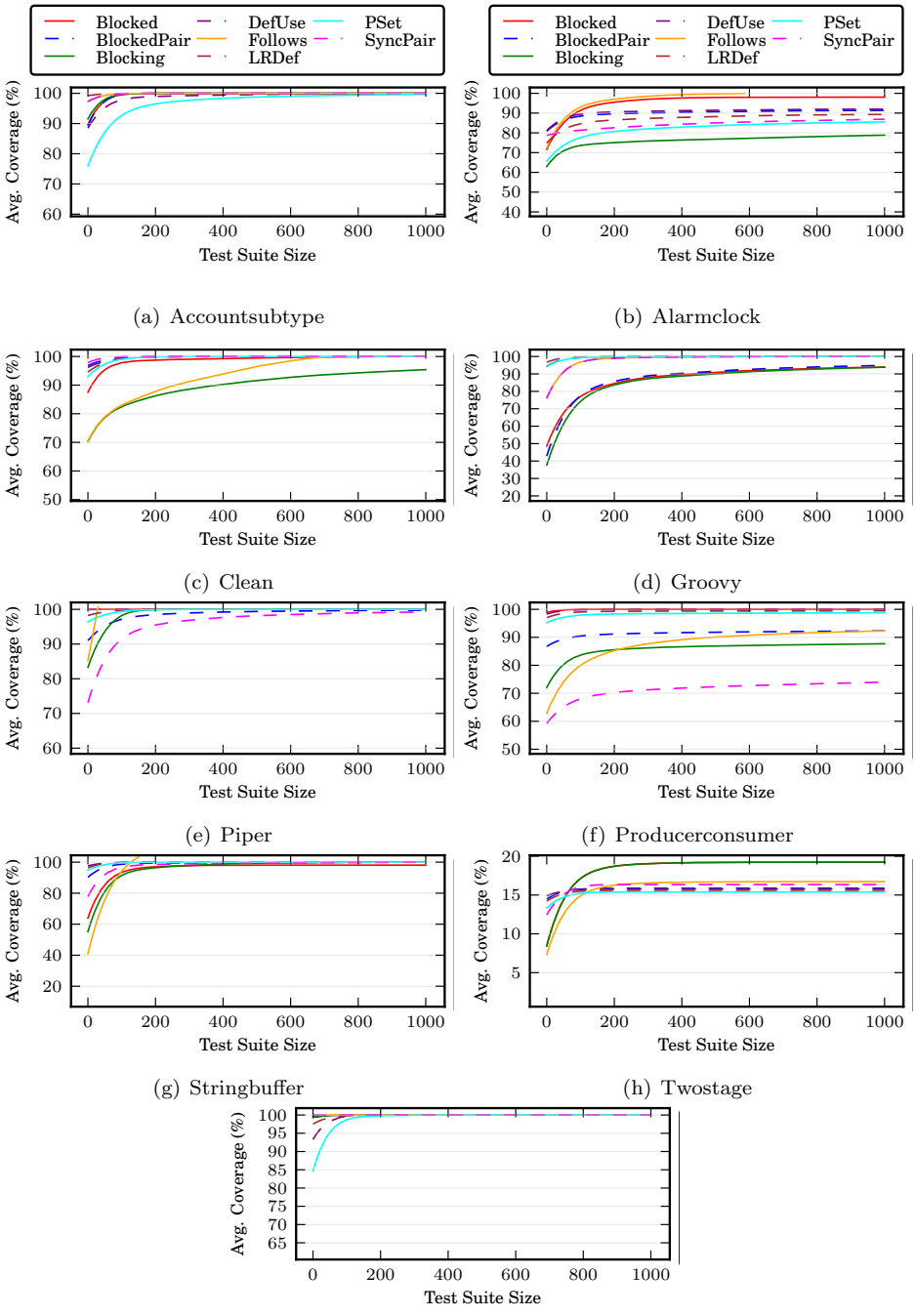
Case 1: for a data write statement of l_1 *and a data read statement of* l_2*: there exist an action* $\sigma[i]$ *and an action* $\sigma[j]$ $(i < j)$ *such that* $\texttt{loc}(\sigma[j]) = l_2 \wedge \texttt{operator}(\sigma[j]) = \texttt{read} \wedge i = \texttt{last-write}_E(\sigma[j]) \wedge \texttt{loc}(\sigma[i]) = l_1 \wedge \texttt{thread}(\sigma[i]) \neq \texttt{thread}(\sigma[j])$

Case 2: for a data write statement of l_1 *and a data write statement of* l_2*: there exist an action* $\sigma[i]$ *and an action* $\sigma[j]$ $(i < j)$ *such that* $\texttt{loc}(\sigma[j]) = l_2 \wedge \texttt{operator}(\sigma[j]) = \texttt{write} \wedge i = \texttt{last-write}_E(\sigma[j]) \wedge \texttt{loc}(\sigma[i]) = l_1 \wedge \texttt{thread}(\sigma[i]) \neq \texttt{thread}(\sigma[j])$

Case 3: for a data read statement of l_1 *and a data write statement of* l_2*: there exist an action* $\sigma[i]$ *and an action* $\sigma[j]$ $(i < j)$ *such that* $\texttt{loc}(\sigma[i]) = l_1 \wedge \texttt{operator}(\sigma[i]) = \texttt{read} \wedge j = \texttt{next-lock}_E(\texttt{i},\texttt{operand}(\sigma[i])) \wedge \nexists k\ (i < k < j).(\ \texttt{operator}(\sigma[k]) = \texttt{read} \wedge \texttt{operand}(\sigma[k]) = \texttt{operand}(\sigma[i])\ )$

Chapter B. Complete Experiment Result Data of Empirical Evaluation on Concurrency Coverage Metrics

This section present the full results (discussed in Section 3.3) for all study objects.



(a) Accountsubtype

(b) Alarmclock

(c) Clean

(d) Groovy

(e) Piper

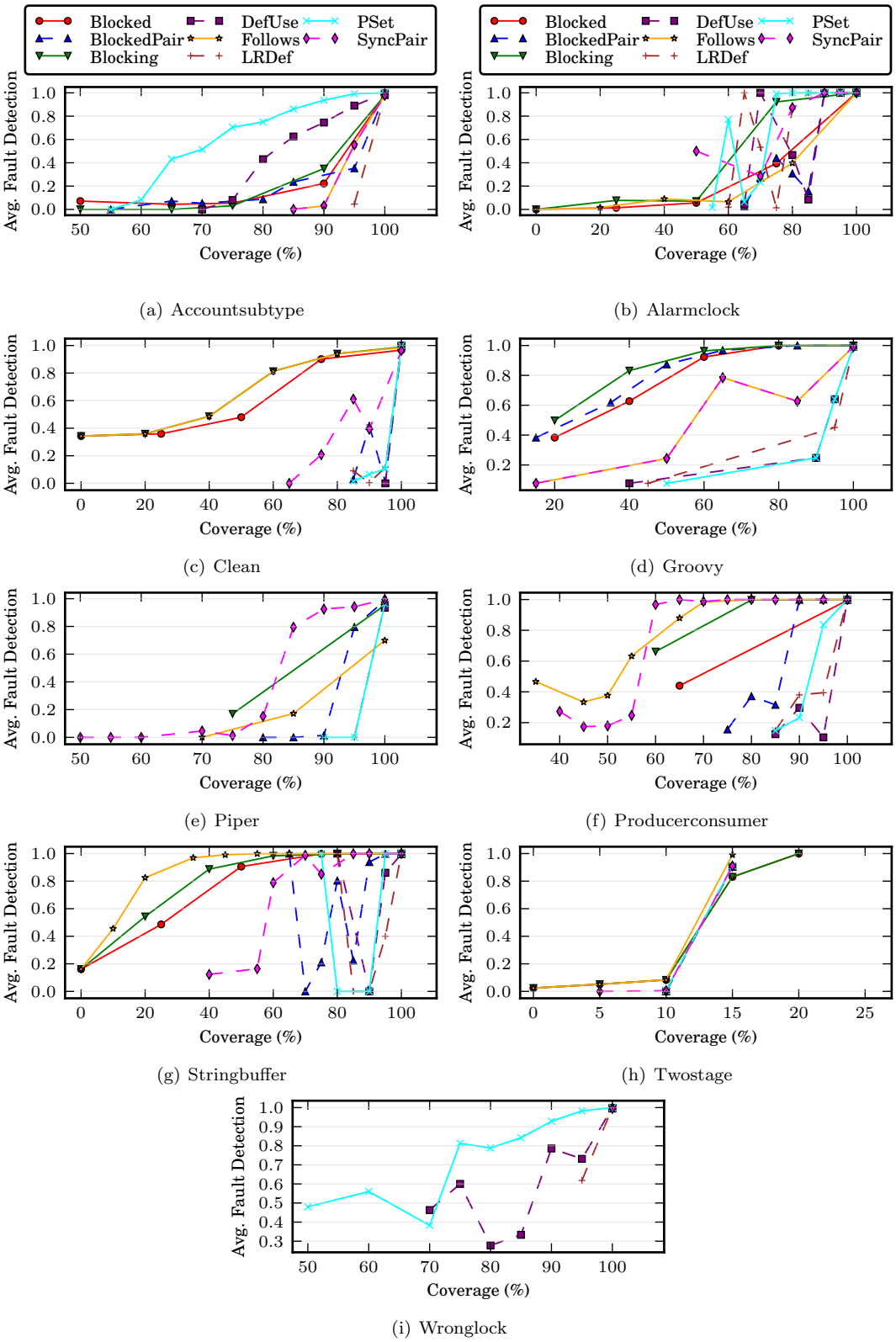
(f) Producerconsumer

(g) Stringbuffer

(h) Twostage

(i) Wronglock

Figure B.1: Size versus coverage, all single fault objects



(a) Accountsubtype

(b) Alarmclock

(c) Clean

(d) Groovy

(e) Piper

(f) Producerconsumer

(g) Stringbuffer

(h) Twostage

(i) Wronglock

Figure B.2: Coverage versus fault detection effectiveness, all single fault objects

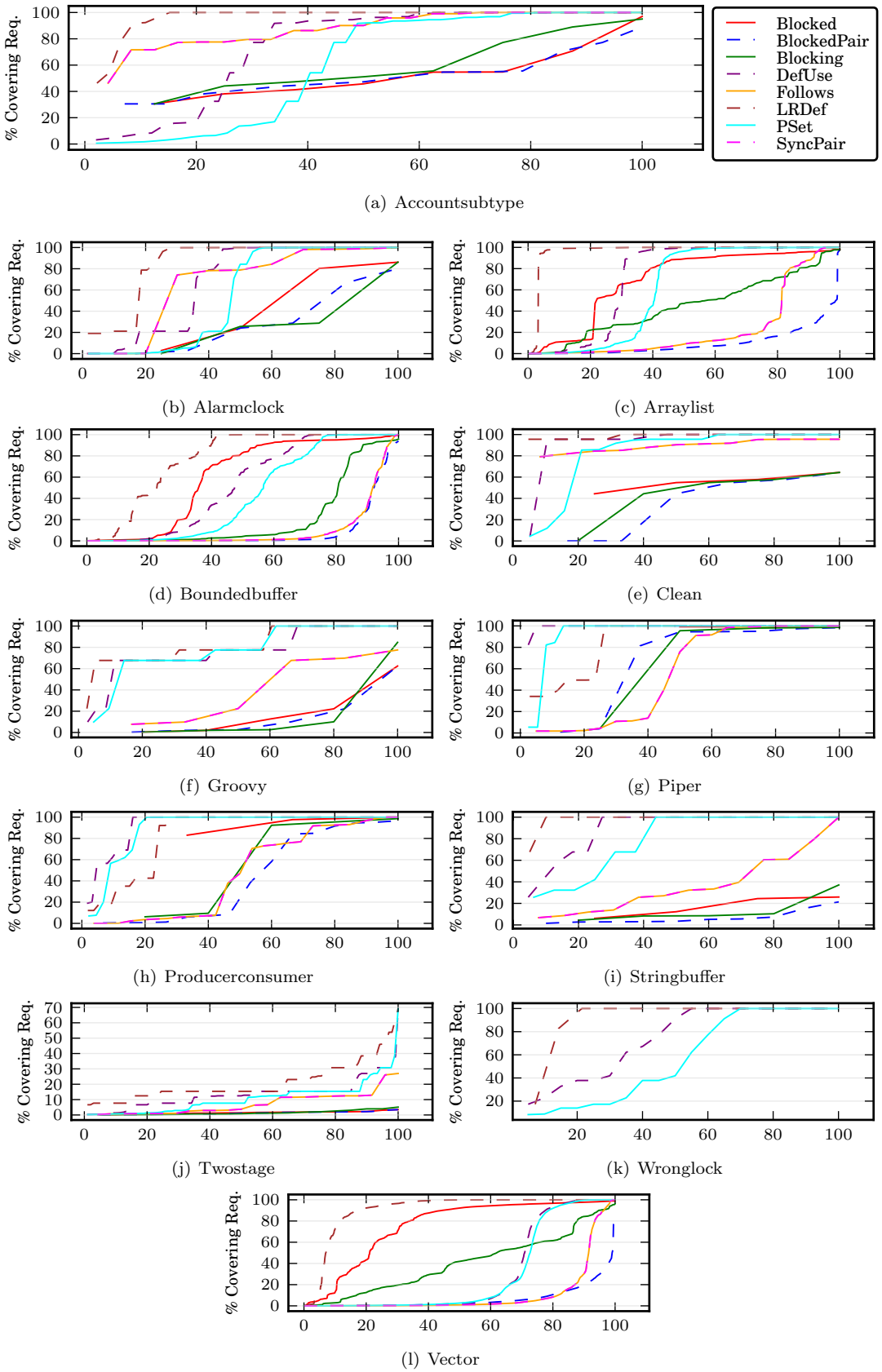


Figure B.3: Percentage of test executions covering test requirements, sorted, all single fault and mutation objects

Summary

Effective and Efficient Test Generation for Multithreaded Programs Using Concurrency Coverage Metrics

오늘날 많은 소프트웨어는 멀티코어 하드웨어를 효과적으로 활용할 수 있는 멀티쓰레드 프로그램(multithreaded program) 형태로 개발되고 있다. 멀티쓰레드 프로그램 개발의 난점 중 하나는 기존의 소프트웨어 테스팅(software testing) 방법이 멀테쓰레드 프로그램의 동시성 오류 검출과 동작정확성 검증에 효과적이지 않다는 점이다. 프로그램 입력 값에 의해서만 동작이 결정되는 비 멀티쓰레드 프로그램(단일쓰레드 프로그램)과 달리, 멀티쓰레드 프로그램의 동작은 입력 값뿐만 아니라 쓰레드 스케쥴(thread schedule), 즉 쓰레드 간 실행순서에 의해서도 영향을 받는다. 일반적인 멀티쓰레드 프로그램의 경우, 쓰레드 스케쥴이 실행 시점에 비결정적(non-deterministic) 쓰레드 스케쥴러에 의해 결정되기 때문에, 규모가 작은 프로그램에 대해서조차 발생 가능한 쓰레드 스케쥴이 극심히 많고, 별도의 장치 없이 쓰레드 스케쥴의 의도적 생성이 어렵기 때문에, 기존의 비 멀티쓰레드 프로그램을 대상으로 한 테스팅 방법으로는 효과적이고 효율적인 테스팅 수행이 어렵다. 멀티쓰레드 프로그램의 오류검출과 동작정확도 검증을 위해 현재까지 개발된 여러 기법은, 분석 능력이 제한적이거나 검증 대상 프로그램 크기에 대한 확장성이 낮아 실제 소프트웨어 개발에서 실용성이 낮은 실정이다.

본 논문은 동시성 커버리지 메트릭(concurrency coverage metric)을 활용하여 멀티쓰레드 프로그램을 효과적이고 효율적으로 테스팅하는 자동 테스팅 기법을 제안한다. 동시성 커버리지 메트릭은, 현재 널리 쓰이는 분기/구문 커버리지 메트릭과 유사하게, 검증 대상 프로그램에서 대한 테스팅 조건을 생성함으로써 멀티쓰레드 프로그램의 체계적인 테스팅을 지원하고자 제안된 방법론이다. 반면, 동시성 커버리지 메트릭이 실제 소프트웨어 테스팅에서 어느 정도 효용성을 제공하는 지 실증적으로 입증되지 않았으며, 그동안 동시성 커버리지 메트릭을 활용한 자동 테스팅 기법도 제한적인 수준이었다. 본 논문은 우선, 동시성 커버리지 메트릭이 테스트 메트릭으로서 멀티쓰레드 프로그램 테스팅에 효과적인 기능을 제공하는지를 실험적 방법으로 검토하였다. 여러 멀티쓰레드 프로그램을 이용한 실험 결과에 따르면, 현재 제안된 대부분의 동시성 커버리지 메트릭은 멀티쓰레드 프로그램 테스팅의 오류 검출능력을 추정하고 유용한 테스트 생성 지표를 제공하는데 효과적인 기능을 제공한다. 본 논문은 두 번째로, 테스팅 과정에서 높은 동시성 커버리지를 단시간에 달성하는 쓰레드 스케쥴 생성 알고리즘을 제시하고, 이를 기반으로 한 자동 테스팅 기법을 소개한다. 본 논문이 제시한 자동 테스팅 기법은, 기존에 제안된 동시성 커버리지 메트릭의 한계점을 개선한 새로운 메트릭인 조합적 동시성 커버리지 메트릭(combinatorial concurrency coverage metric)를 활용한다. 본 논문이 제시한 자동 테스팅 기법을 기존의 멀티쓰레드 프로그램 테스팅 기법과 비교한 실험 결과에 따르면, 본 논문의 기법이 기존 기법보다 향상된 멀티쓰레드 프로그램 오류 검출 효용성과 효율성을 달성함을 알 수 있다. 마지막으로, 본 논문은 동시성 커버리지 메트릭을 활용하여 멀티쓰레드 프로그램에 대해 효과적으로 회기 테스팅(regression testing)을 수행하는 동시성 커버리지 기반 회기 테스팅 기법을 제시한다. 본 논문이 제시한 기법을 기존 기법과 비교한 실험결과에 따르면, 동시성 커버리지 기반 회기 테스팅 기법은 멀티쓰레드 프로그램 수정 과정에서 발생하는 동시성 회기 오류를 기존 기법보다 효과적이고 효율적으로 검출한다.

Acknowledgement

I praise my Lord Jesus Christ for his presence in my Ph.D study. I thank that His words teach me the enduring mercy on this son of little faith and his good wills in the everlasting love all through the years. His faithfulness protected me and made me proceed. I hope that his lead always be in my life.

I would like to express my sincere appreciation to my advisor, Prof. Moonzoo Kim for his guidance, support, trust, care and prayer. He knows me and offers me best opportunity at every step of my Ph.D course. His enthusiastic and professional attitude to research and education will keep teach me along my career. I also thank my Ph.D committee members, Prof. Sukyoung Ryu, Prof. Taisook Han, Prof. Jaehyuk Huh at KAIST, and Prof. Chao Wang at Virginia Tech. With their valuable comments, I could improve my dissertation.

I was pleased to meet and work with talented researchers through my Ph.D study. Prof. Gregg Rothermel teaches me empirical software engineering in his visiting at KAIST, which promotes me to start empirical investigations into the concurrency coverage metrics. Through research collaboration and discussion, I had learned many aspects of research from Dr. Matt Staats. Thanks for being a great co-worker and many cups of coffee. I thank Dr. Sangmin Park for his collaboration at the initial work of coverage-guided multithreaded program testing. I appreciate my research partners, Mr. Jaemin Ahn and Mr. Yongbae Park for their devotional collaborations on the experiments. I was fortunate to work with great colleagues in Software Testing and Verification Group at KAIST. Especially, I am grateful to have Mr. Yunho Kim as the running mate of my graduate study. Without his constant encouragement and help, I cannot imagine how would I pass through all steps of my research.

Lastly but most importantly, I would express my deepest appreciation to my lovely wife, Eunjeong Lee and my son, Yeh-June Hong; unspeakable love, trust and endurance always fill my heart and power. I am so proud of my family, who always support me in everything. Special thanks to my mother and father, and my parent in law for their love and encouragements. I also thank my brothers and sisters in Jesus at Daedukhanbit Church for the care and prayer.

This work is supported in part by the National Research Foundation of Korea (NRF) Mid-career Research Program (NRF-2012R1A2A2A01046172) and the ITRC (Information Technology Research Center) Support Program supervised by the NIPA (National IT Industry Promotion Agency), funded by the Ministry of Science, ICT and Future Planning (MSIP), Korea.

이 력 서

이 름 : 홍신

생 년 월 일 : 1985년 6월 18일

주 소 : 대전시 유성구 농대로 41 궁동 카이스트 아파트 102동 405호

E-mail 주 소 : hongshin@kaist.ac.kr

학 력

2001. 3. – 2003. 2. 부산과학고등학교 졸업 (現 Korea Science Academy of KAIST)

2003. 3. – 2007. 2. KAIST 전산학과 학사 (B.S.)

2007. 3. – 2010. 2. KAIST 전산학과 석사 (M.S.)

2011. 2. – 현재 KAIST 전산학과 박사과정 (Ph.D)

경 력

2007. 3. – 2008. 12. KAIST 전산학과 조교

2010. 3. – 2011. 1. KAIST 전산학과 연구원

2011. 2. – 2014. 12. KAIST 전산학과 조교

학회활동

1. Y. Park, **S. Hong**, M. Kim, D. Lee, and J. Cho, Systematic Testing of Reactive Software with Non-deterministic Events: A Case Study on LG Electric Oven, International Conference on Software Engineering (ICSE), Software Engineering In Practice Track (SEIP), May 2015.
2. **S. Hong**, Y. Park, M. Kim, Detecting Concurrency Errors in Client-side JavaScript Web Applications, IEEE International Conference on Software Testing, Verification and Validation (ICST), Mar 2014.
3. **S. Hong**, M. Staats, J. Ahn, M. Kim, and G. Rothermel, The Impact of Concurrent Coverage Metrics on Testing Effectiveness, IEEE International Conference on Software Testing, Verification and Validation (ICST), Mar 2013.
4. M. Staats, **S. Hong**, M. Kim, and G. Rothermel, Understanding User Understanding: Determining Correctness of Generated Program Invariants, Intl. Symp. on Software Testing and Analysis (ISSTA), Jul 2012.
5. **S. Hong**, J. Ahn, S. Park, M. Kim, and M. J. Harrold, Testing Concurrent Programs to Achieve High Synchronization Coverage, Intl. Symp. on Software Testing and Analysis (ISSTA), Jul 2012.

6. M. Kim, **S. Hong**, C.Hong and T.Kim, Model-based Kernel Testing for Concurrency Bugs through Counter Example Replay, Model-based Testing (ENTCS vol 253, no 2), Mar 2009.

연 구 업 적

1. **S. Hong**, M. Staats, J. Ahn, M. Kim, G. Rothermel, Are Concurrency Coverage Metrics Effective for Testing: A Comprehensive Empirical Investigation, Journal of Software Testing, Verification and Reliability (STVR), volume 25, issue 4, pages 334-370, Jun 2015.
2. **S. Hong** and M. Kim, A survey of race bug detection techniques for multithreaded programs, Journal of Software Testing, Verification and Reliability (STVR), volume 25, issue 3, pages 191-217, May 2015.
3. **S. Hong** and M. Kim, Effective Pattern-driven Concurrency Bug Detection for Operating Systems, Journal of Systems and Software (JSS), volume 86, issue 2, pages 377-388, Feb 2013.