UC Berkeley
UC Berkeley Electronic Theses and Dissertations

Title
Predictive and Programmable Testing of Concurrent and Cloud Systems

Permalink
https://escholarship.org/uc/item/0161k90g

Author
Joshi, Pallavi

Publication Date
2012

Peer reviewed|Thesis/dissertation

eScholarship.org Powered by the California Digital Library
University of California

Predictive and Programmable Testing of Concurrent and Cloud Systems

by

Pallavi Joshi

A dissertation submitted in partial satisfaction of the
requirements for the degree of
Doctor of Philosophy

in

Computer Science

in the

Graduate Division

of the

University of California, Berkeley

Committee in charge:

Professor Koushik Sen, Chair
Professor George Necula
Professor Leo A. Harrington

Fall 2012

Predictive and Programmable Testing of Concurrent and Cloud Systems

Copyright 2012
by
Pallavi Joshi

Abstract

Predictive and Programmable Testing of Concurrent and Cloud Systems

by

Pallavi Joshi

Doctor of Philosophy in Computer Science

University of California, Berkeley

Professor Koushik Sen, Chair

Today's software systems often have poor reliability. In addition to losses of billions, software defects are responsible for a number of serious injuries and deaths in transportation accidents, medical treatments, and defense operations. The situation is getting worse with concurrency and distributed computing becoming integral parts of many real-world software systems. The non-determinism in concurrent and distributed systems and the unreliability of the hardware environment in which they operate can result in defects that are hard to find and understand.

In this thesis, we have developed tools and techniques to augment testing to enable it to quickly find and reproduce important bugs in concurrent and distributed systems. Our techniques are based on the following two key ideas: (i) use program analysis to increase coverage by predicting bugs that could have occurred in "nearby" program executions, and (ii) provide programming abstractions to enable testers to easily express their insights to guide testing towards those executions that are more likely to exhibit bugs or help achieve testing objectives without having any knowledge about the underlying testing process. The tools that we have built have found many serious bugs in large real-world software systems (e.g. Jigsaw web server, JDK, JGroups, and Hadoop File System).

In the first part of the thesis, we describe how we can predict and confirm bugs in the executions of concurrent systems that did not show up during testing but that could have shown up had the program under consideration executed under different thread schedules. This improves the coverage of testing, and helps find corner-case bugs that are unlikely to be discovered during traditional testing. We have built *predictive testing* tools to find different classes of serious bugs like deadlocks, hangs, and typestate errors in concurrent systems.

In the second part of the thesis, we investigate how we can improve the efficiency of testing of distributed cloud systems by letting testers guide testing towards the executions that are interesting to them. For example, a tester might want to test those executions that are more likely to be erroneous or that are more likely to help her achieve her testing objectives. We have built tools and frameworks that enable testers to easily express their knowledge and intuition to guide testing without having any knowledge about the underlying

testing process. We have investigated *programmable testing* tools in the context of testing of large-scale distributed systems.

To Mummy and Daddy

Contents

Contents ii

List of Figures v

List of Tables viii

I Testing Multithreaded Programs 1

1 Introduction 2

2 Overview 5

3 Background Definitions 10
- 3.1 Events . . . 11
- 3.2 Multithreaded execution state . . . 14

4 Resource Deadlocks 15
- 4.1 Predictive testing for resource deadlocks . . . 15
- 4.2 Overview . . . 16
- 4.3 iGoodlock . . . 18

5 Generalized Deadlocks 23
- 5.1 Rationale . . . 24
- 5.2 Overview . . . 26
- 5.3 Predictive Testing for Communication Deadlocks . . . 31
- 5.4 CheckMate and iGoodlock . . . 37

6 Typestate Errors 38
- 6.1 Overview . . . 40
- 6.2 Predictive Testing for Typestate Errors . . . 42

7 Confirming Predicted Errors 49

7.1 Overview . . . 49
7.2 Confirming Resource Deadlocks . . . 51
7.3 Object abstractions . . . 54
7.4 Optimization: avoiding another potential cause for thrashing . . . 57
7.5 Confirming other concurrency errors . . . 58

8 Implementation **60**
8.1 The instrumentor . . . 60
8.2 Implementing analyses . . . 61

9 Evaluation **63**
9.1 Resource deadlocks . . . 63
9.2 Communication deadlocks . . . 66
9.3 Typestate errors . . . 70
9.4 Confirming Resource Deadlocks . . . 72

10 Other Related Work **77**
10.1 Predictive testing . . . 77
10.2 Static analysis and Model checking . . . 79

II Testing Distributed Systems **80**

11 Introduction **81**

12 Overview **84**
12.1 Example 1: Leader Election . . . 84
12.2 Example 2: Distributed File System . . . 86

13 Background Definitions **90**
13.1 Execution related abstractions . . . 90
13.2 Profiling executions . . . 92

14 Programmable Model Checking **93**
14.1 Overall Architecture . . . 93
14.2 MC Facilitator . . . 95
14.3 MC Driver . . . 96
14.4 MC Engine . . . 99

15 Programmable Failure Injection **104**
15.1 Test Workflow . . . 104
15.2 FI Engine . . . 105
15.3 FI Driver . . . 106

15.4 Test Workflow Algorithm 108
15.5 Crafting Pruning Policies 110

16 Implementation **121**
16.1 The instrumentor 121
16.2 The policy framework 124
16.3 Tool complexity 126

17 Evaluation **127**
17.1 Effectiveness of Policies 127
17.2 Ease of writing policies 133
17.3 Bugs found 134

18 Other related work **135**

19 Conclusion **138**

Bibliography **140**

List of Figures

1.1 Coverage: Traditional testing versus Predictive testing 3

2.1 Thread interleaving causing no deadlock 6
2.2 Thread interleaving causing deadlock 6
2.3 Trace of operations (or events) obtained from observing the execution in Figure 2.1 8

4.1 Example of a resource deadlock 16
4.2 Simple example of a deadlock 17

5.1 Recommended condition variable usage. We use synch to abbreviate synchronized. 24
5.2 Deadlock due to missed notification. 26
5.3 Deadlock involving both locks and condition variables. 26
5.4 An example with a communication deadlock. 28
5.5 Non-deadlocking interleaving for example in Figure 5.4. 29
5.6 Deadlocked interleaving for example in Figure 5.4. 29
5.7 Trace program generated by observing the execution of the interleaving in Figure 5.5 of the example in Figure 5.4. 30
5.8 Definition of class `ConditionAnnotation`. 32
5.9 Definition of class `AddLinesToTraceProgram` that is used by class `ConditionAnnotation` (Figure 5.8) and Algorithm 2. 33

6.1 Example of a typestate error: An IOException is thrown when getOutputStream() is invoked on a closed Socket object 39
6.2 The likely typestate specification automaton learnt for Socket objects 40
6.3 The multithreaded computation lattice for Figure 6.1 41

7.1 Confirming a Resource Deadlock 50

9.1 Deadlock in Jigsaw 65
9.2 Deadlock in `groovy`. 69
9.3 Deadlock in `jigsaw`. 70
9.4 Typestate error in `weblech` 72
9.5 Typestate automaton inferred for DownloadQueue 72

9.6 Performance and effectiveness of variations of DeadlockFuzzer 74

12.1 Different possible orderings of messages at Node 3 in the initial stage of the leader election protocol. V stands for "Vote". 85
12.2 Reorder votes v_1 and v_2 only if they are different and greater than the current leader at the receiver node 86
12.3 I/Os executing in two nodes in a distributed file system 87
12.4 Inject crashes before disk I/Os 88

14.1 Architecture of PCheck. **1**: Blocked events, **2**: Model checking decision, **3**: Policies, and **4**: Abstractions 94
14.2 Gossip protocol: Reorder message receive events e_1 and e_2 that are final acks for gossip requests 97
14.3 Client write protocol: Reorder message receive events e_1 and e_2 that are acks for different write requests 97
14.4 Leader election protocol: Reorder message receive events e_1 and e_2 that have votes that are greater than the current leader at the receiver 98
14.5 An example execution tree. The MC Engine performs depth-first search with dynamic partial-order reduction on execution trees. 101

15.1 PreFail's Architecture. The figure shows the separation of PreFail into failure-injection engine and driver. 105
15.2 PreFail's Test Workflow 105
15.3 Setup-stage filter: Return true if all `fits` $(ft_1, fp_1), \ldots, (ft_n, fp_n)$ in fs correspond to a crash within the `setup` function. 107
15.4 Recovery path cluster: Cluster two failure sequences if their last `fits` are the same and their prefixes (that exclude the last `fits`) result in the same recovery path. 108
15.5 HDFS Write Protocol 111
15.6 Ignore nodes cluster: Return true if two failure sequences have the same failures with the same contexts not considering the nodes in which they occur. 111
15.7 New source location filter: Return true if the source location of the last `fip` has not been explored 112
15.8 Source location cluster: Return true if the `fips` in the last `fits` have the same source location. 112
15.9 Obtaining Recovery Path FIPs 114
15.10 Considering only source locations to judge equivalence of recovery paths 115
15.11 Recovery Classes of HDFS Write Protocol 115
15.12 Equivalent-recovery clustering: Cluster two failure sequences if their prefixes result in the same recovery path and their last `fits` are the same 116
15.13 Generic crash optimization: The function accepts a failure sequence if all crashes in the sequence are injected before write I/Os 117

15.14 Crash optimization for network writes 118
15.15 Network failure optimization 119

16.1 Architecture of an instrumented system 122

17.1 Policies for ZK-LE1 130
17.2 Policies for ZK-LE2 130
17.3 #Experiments run with different coverage-based policies 131

List of Tables

9.1 Experimental results for resource deadlocks 64
9.2 Experimental results for CheckMate 67
9.3 Execution time for typestate checking 71
9.4 Experimental results for DeadlockFuzzer. (Context + 2nd Abstraction + Yield optimization) 73

13.1 Event abstraction 91
13.2 Failure-Injection Point (`fip`) and Failure-Injection Task (`fit`) 91

15.1 HDFS Write Recovery 113

17.1 Branch coverage with policies. Here, N = 1, that is, we allow reordering of at most one pair of events in each execution 128
17.2 Effect of increasing N. Here, N = 2, that is, we allow reordering of at most two pairs of events in each execution 131
17.3 #Bugs found. (*) implies that these are the same bugs (*i.e.*, bugs in 2-failure cases often appear again in 3-failure cases). 132
17.4 Benefits of Optimization-based Policies. (*) These write workloads do not perform any disk read, and thus the optimization does not work here. 133

Acknowledgments

This thesis would not at all have been possible without the guidance of my advisor, Koushik Sen. Over the last six years, Koushik has patiently mentored me and has taught me what it takes to do good research. Under his guidance, I have learnt the art of choosing the right research problems, writing strong papers, and giving good talks. He has always encouraged me to find my own research problems, and has whole-heartedly supported me in pursuing the research directions that I have wanted to explore. He has always been available to meet whenever I have needed his help. His casual check-ins and "what's up" have often led to hour long discussions that have helped me look at a problem differently or have helped me find a better solution for a problem that I had not thought of before. I will always be in debt of Koushik for his guidance, patience, and kindness.

Mayur Naik and Haryadi Gunawi have been like my second advisors in two significant projects that I have worked on for this thesis. I am grateful to Mayur for giving me the opportunity to work as an intern with him. From working with Mayur, I have learnt how to keep making regular research progress without getting stuck with a problem for too long. I am also thankful to him for all his support during my job search. Haryadi has taught me everything I know about systems research. He has always patiently answered all my questions regarding implementation of distributed systems, and has always been there to guide me and brainstorm with me.

I want to thank George Necula for kindly agreeing to serve on my qualifying exam committee and dissertation committee. The discussions with George, and his questions and feedback have greatly improved this thesis. I am also grateful to George for his advice and guidance during my job search. George as a teacher has been an inspiration for me. His course on the principles of programming languages is one of the best courses that I have ever taken, and is also what got me interested in the area of programming languages and software engineering. I also want to thank Ras Bodik and Leo Harrington for their suggestions and comments that have helped shape this thesis.

Thank you to my wonderful colleagues, Jacob, Chang-Seo, Joel, Tayfun, Christos, and Philip for always providing me with detailed and constructive feedback on my drafts and practice talks. Thank you to Raluca for making it a lot more fun to work on course projects and to prepare for the preliminary exam. More than that, Raluca has been a great friend all these years, and I will always cherish our road trips and movie sessions together. Thank you to the OSQ community for providing me with a platform to present my research regularly. Roxana, Tammy, and La Shana took care of all my administrative requests, and made it much easier and quicker to deal with filling forms and doing paperwork. Thank you to all of them.

Life would not have been as much fun as it has been had it not been for my friends here in Berkeley. Shashank, Rahul, Shradha, Nandini, Yamini, Deepan, Narayanan, Mohit, Siddharth, Raluca, Arun – thank you for all the amazing times together, for celebrating the highs with me, and for being there during the lows. I would also like to thank all my housemates over the last six years for putting up with me and my idiosyncrasies.

Last but not the least, I want to thank my family. My brother has always encouraged me to dream the impossible and to do my best to achieve that dream. I always count on him for the right advice in any situation. My parents have always believed in me, and have encouraged me in all my academic pursuits. Everything that I have achieved in my life is because of their support, love, and care. I wouldn't have finished this thesis had it not been for their continuous encouragement, motivation, and advice.

Part I

Testing Multithreaded Programs

Chapter 1

Introduction

For decades, the Moore's law held true, and the performance of single-core processors almost doubled every eighteen months. But, in recent years, because of the limitations imposed by physics, we have not been able to continue with the same rate of improvement in performance in single-core processors. As a result, the processor manufacturing world has shifted its focus towards multi-core processors that pack multiple CPUs on a single chip. With multi-core processors, we can potentially execute different parts of a software system in parallel, and thus improve the performance of the software system.

One of the main challenges of using multi-core processors is writing correct multithreaded programs that can execute different threads of computation in parallel on different cores. Multithreaded programs can have an enormous amount of interaction going on between different simultaneous threads that can get mind-boggling and tedious for a programmer to keep track of and reason about. Thus, programmers can often unintentionally allow incorrect interaction between threads in their programs. Such errors are difficult to detect because they show up only under very specific thread schedules. Thus, even if a particular execution of a multithreaded program does not exhibit an error, it does not mean that the program does not have an error for the input used. There might be an error that could have occurred had the threads interleaved in a different manner.

The common way to test multithreaded programs is to stress those programs, that is, to operate those programs under extreme conditions like high concurrency and heavy load so that the rare erroneous thread interleavings become more likely to show up. For example, to stress a program, one can spawn a large number of concurrent threads, or run the program for days and weeks, or add randomness and delays in the program to perturb the program execution unexpectedly. Stress testing might lead to the exhibition of rare interleavings, but it is quite ad-hoc and does not systematically try to find the rare erroneous thread interleavings. In our work, we have tried to address this drawback by adding in the right intelligence to testing techniques that helps them to quickly predict and confirm bugs in the incorrect thread interleavings that did not show up during program execution. The intelligence is provided by an appropriate program analysis that is suitable for the type of bugs that we are interested in. The program analysis is performed in the background,

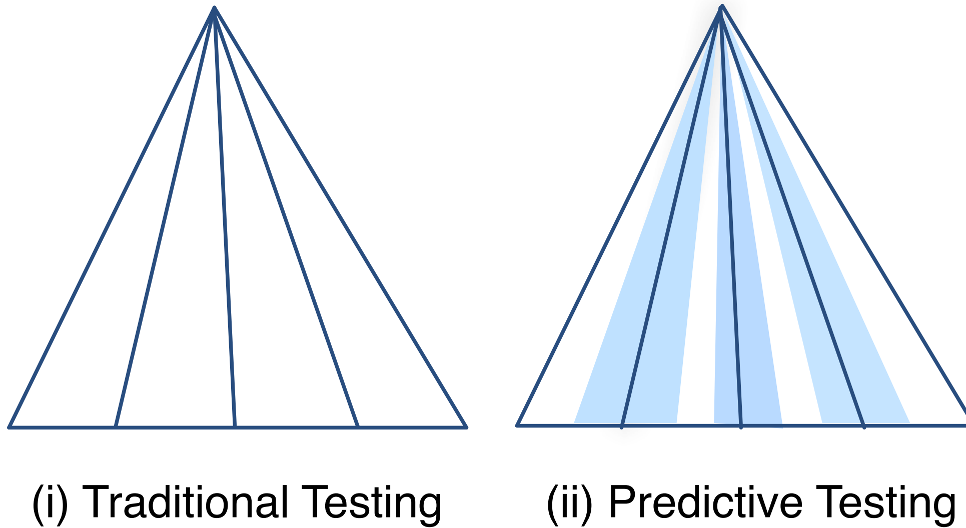


Figure 1.1: Coverage: Traditional testing versus Predictive testing

and thus the tester does not have to bother about learning any details about the analysis, and can continue with testing as she is accustomed to. We call this predictive testing. Predictive testing is not new. There has been a significant amount of work on predictive testing [93, 24, 108, 86, 39, 12, 6], but most of this work has focused only on prediction of errors. Predicted errors however may not always be real errors as the predictive techniques might use approximation that might result in the reporting of false positives (predicted errors that are not real errors). In our work, in addition to predicting errors, we also try to automatically confirm which of the predicted errors are real errors.

Figure 1.1 illustrates how predictive testing analyzes "nearby" thread interleavings that did not show up during traditional testing, and uses appropriate program analysis in the background to quickly predict and confirm bugs in those interleavings. A "nearby" interleaving is one that is similar to one of the interleavings observed during normal testing, but has some thread synchronization operations executing in a different order. The triangle on the left in Figure 1.1 represents the set of all thread interleavings that were observed during normal testing. For each interleaving that was observed, predictive testing uses the right program analysis to find the bugs of the kind that we are interested in in "nearby" interleavings. The shaded region around each thread interleaving in the triangle in the right in Figure 1.1 represents the "nearby" interleavings that predictive testing considers. Since predictive testing analyzes "nearby" interleavings without actually executing them, it can only find potential bugs that can occur in those interleavings. A potential bug that is found might or might not correspond to a real bug that can occur during an actual program execution since predictive testing typically approximates "nearby" interleavings to find bugs

in them. The approximation might result in some false positives (reported errors that are not real errors). Therefore, after predictive testing reports potential bugs, we also try to automatically confirm if a reported bug is a real bug or not. Specifically, we actively control the thread scheduler during program execution to mimic the thread interleaving that can potentially lead to the reported bug.

There are various kinds of errors or bugs that can happen in multithreaded programs, *e.g.* data races, deadlocks, atomicity violations, and typestate errors. In this thesis, we have focused on deadlocks and typestate errors, and will be presenting predictive testing techniques for finding them in subsequent chapters. Even though the predictive testing techniques are designed for finding deadlocks and typestate errors, different parts of those techniques can be applied to finding other classes of bugs. We will be pointing out the ideas that can be generalized to find other kinds of bugs as we explain the techniques.

In the subsequent chapters, we first give an overview of how predictive testing works with the help of an example (Chapter 2). We then provide background definitions in Chapter 3, and use them when explaining predictive testing techniques for deadlocks (Chapters 4 and 5) and typestate errors (Chapter 6). We describe how predicted errors can be confirmed by actively controlling the thread scheduler in Chapter 7. Finally, we provide the implementation and a detailed evaluation of all our analyses (Chapters 8 and 9), and conclude by comparing our work against related work in Chapter 10.

Chapter 2

Overview

In this chapter, we provide a high-level overview of how predictive testing predicts and confirms bugs in thread interleavings that did not show up during normal testing. Consider the interleaving of the two threads in Figure 2.1. The first thread acquires locks on the objects L1 and L2, and subsequently releases the two locks. It then acquires a lock on a different object L3, and modifies the value of the variable SV. The variable SV is shared between the two threads, and therefore the threads use locks to synchronize their accesses to SV. The thread subsequently releases the lock on L3. All the while the first thread is executing, the second thread is executing a long running method longM(). After it is done with executing longM(), it reads the value of SV into a local variable T while holding the lock L3. After that it also acquires locks on L1 and L2, and releases them subsequently. Note that the second thread acquires locks on L1 and L2 in reverse order, that is, it first acquires the lock on L2 and then the lock on L1. The two threads might perform some operations while holding the locks on L1 and L2, but we do not show the operations here to keep the example simple.

There is no deadlock in the interleaving shown in Figure 2.1, that is, neither of the threads gets blocked forever during execution. But, there is another not so likely interleaving that could have resulted in a deadlock. In the deadlocking interleaving (Figure 2.2), the first thread acquires the lock on L1, but before it can acquire the lock on L2, the second thread finishes the execution of longM(), reads SV, and acquires the lock on L2. The first thread now gets blocked while trying to acquire the lock on L2 as the second thread has not yet released it. The second thread also gets blocked when trying to acquire the lock on L1 as the first thread is still holding it. As a result, there is a deadlock. The deadlocking interleaving is unlikely to show up during normal execution because the execution of longM() takes so long that the first thread would likely acquire and release all the locks before the second thread can start acquiring locks. But, nevertheless, the erroneous interleaving is feasible, and the deadlock should be detected and fixed in the program.

If we test the program that has the two threads by just executing it repeatedly, it is unlikely that the deadlocking interleaving would show up. Thus, traditional testing might miss the deadlock in the program. Even if the deadlock does not show up in any program

```
Thread T1                                   Thread T2

lock(L1);                                    longM();
lock(L2);
unlock(L2);
unlock(L1);

lock(L3);
SV = 100;
unlock(L3);
                                             lock(L3);
                                             T = SV;
                                             unlock(L3);

                                             lock(L2);
                                             lock(L1);
                                             unlock(L1);
                                             unlock(L2);
```

Figure 2.1: Thread interleaving causing no deadlock

```
Thread T1                                   Thread T2

lock(L1);                                   longM();

                                            lock(L3);
                                            T = SV;
                                            unlock(L3);

                                            lock(L2);
```

Figure 2.2: Thread interleaving causing deadlock

execution, predictive testing can detect the deadlock by analyzing "nearby" thread interleavings. For each execution observed, predictive testing records information about the different operations observed during the execution. It uses this gathered information to predict and confirm errors in thread interleavings that did not show up during normal testing. For example, let us say that while observing the interleaving in Figure 2.1, we record information about the thread synchronization operations. Thus, the only operations that we record are the lock acquires and releases. Let LOCK(L, T) denote the acquire of lock on L by thread T, and UNLOCK(L, T) denote the release of lock on L by thread T.

When observing the thread interleaving in Figure 2.1, we can generate the trace of operations or events as shown in Figure 2.3. Note that we do not record any information regarding `longM()` and read and write of SV. This is because method calls and variable accesses are not necessary to find deadlocks. When designing a predictive testing technique for a particular class of errors, one needs to find the set of operations that the technique should keep track of when observing a program execution. The right set of operations would include the thread synchronization operations and other operations that are necessary to be able to detect that class of errors. For example, when finding communication deadlocks (another class of deadlocks explained in Chapter 5), one needs to keep track of lock acquires and releases, and also signals and notifications and the change in values of the conditions on which they depend, and when finding typestate errors (Chapter 6), one needs to record method calls on objects of type of interest in addition to synchronization operations.

There are two main advantages of generating a trace by keeping track of only a subset of relevant operations: (i) the trace is simpler than the actual program execution and thus it is easier and faster to analyze it and find errors for other orderings of events in it, and (ii) throwing away non-relevant operations and inter-thread dependencies enables us to analyze those orderings of events in the trace that would not have been possible had we retained all operations and dependencies. Some of these orderings might correspond to real executions of the program under consideration. For example, had we retained accesses to the shared variable SV in the trace in Figure 2.3, and considered the dependency of the read of SV in the second thread on the write in the first thread, we would not have been able to order the read in the second thread before the write in the first thread and would have thus missed the deadlocking interleaving (Figure 2.2).

After we have generated the trace of events, we can use an appropriate program analysis to find bugs that could have occurred had the events ordered in a different manner. For example, from the trace in Figure 2.3, we can find that the lock acquire and release of L3 and lock acquire of L2 can execute in thread T2 before thread T1 tries to acquire the lock on L2. This is because there is no dependency in the trace that prevents the operations to be executed in this manner. Thus, we can find a potential deadlock by reordering the events in the trace. Since we are tracking only a subset of all operations that occur during execution, we are approximating the execution observed with the event trace that we generate. Thus, when we predict a bug that can occur for a different ordering of events in the trace, that bug might or might not correspond to a real bug that can manifest during an actual program execution. If we just report the bugs that we find by analyzing the event trace, then the

```
LOCK(L1, T1)
LOCK(L2, T1)
UNLOCK(L2, T1)
UNLOCK(L1, T1)
LOCK(L3, T1)
UNLOCK(L3, T1)
LOCK(L3, T2)
UNLOCK(L3, T2)
LOCK(L2, T2)
LOCK(L1, T2)
UNLOCK(L1, T2)
UNLOCK(L2, T2)
```

Figure 2.3: Trace of operations (or events) obtained from observing the execution in Figure 2.1

tester would have to go through all the potential bugs one by one and reason to see if it can occur in a real program execution. The process of reasoning can easily get tedious. Thus, apart from predicting bugs, the predictive testing techniques that we have developed also try to automatically confirm which of the predicted bugs are real bugs.

From the trace in Figure 2.3, we can infer that had the second thread acquired and released the lock on L3, and then acquired the lock on L2 before the first thread could try to acquire the lock on L2, then there would have been a deadlock. Let us say that we want to check if this can happen in a real execution of the program. For this, we execute the program again. But, this time, when the first thread tries to acquire the lock on L2, we pause the thread and let the second thread execute. We want the second thread to acquire the lock on L2 before the first thread tries to acquire the lock. After executing longM(), thread T2 acquires and releases the lock on L3, and then acquires the lock on L2. But, when it tries to acquire the lock on L1, it gets blocked because thread T1 has not released the lock on L1. At this point, if we allow thread T1 to proceed, it gets blocked when acquiring the lock on L2 because thread T2 has not released the lock yet. As a result, both the threads get blocked resulting in a deadlock. Thus, by actively controlling the thread scheduler during program execution, we have forced the execution to follow an interleaving that exhibits the deadlock. We use hints from the analysis of the event trace (*e.g.*, pause the first thread after it has acquired the lock on L1 but before it acquires the lock on L2) to figure out the preemption points during execution where we should pause certain threads to force an interleaving that would likely exhibit an error. Designing the right error confirmation technique involves figuring out the right preemption points for the kind of bugs that we are interested in confirming. For example, for confirming communication deadlocks (Chapter 5), we would need to preempt before lock acquires or releases, signals or notifications, or the

conditions on which the signals or notifications that are involved in the potential deadlocks depend.

In the following chapters, we describe in detail predictive testing techniques that we have designed for deadlocks and typestate errors which are two common classes of bugs in multithreaded programs. Even though the techniques are built for these specific classes of bugs, a significant part of the techniques can be generalized and applied to finding other classes of bugs. We explain which parts of the techniques are generalizable as we describe those techniques.

Chapter 3

Background Definitions

In this chapter, we provide the formalizations and definitions that we use to describe the various predictive techniques later. We formalize various aspects of the multithreaded program execution that we use in our techniques. A multithreaded execution consists of a finite number of threads. We assume that each thread is started only once. At any instant, the execution is in a state s, in which each thread is at a statement. It transitions from one state to another with the execution of a statement by a thread. The initial state is denoted by s_0.

Threads communicate with each other via shared objects. Since simultaneously accessing and modifying objects might lead to inconsistencies in the object states, threads should be correctly synchronized when accessing shared data. This can be done using locks. When a thread acquires a lock, no other thread can acquire the lock until the thread holding it has released it. Thus, if we protect accesses to shared data using locks, then we can ensure that at most a single thread can modify shared data at any point during execution. We consider locks to be re-entrant, i.e., a thread may re-acquire a lock it already holds. A thread must release a lock it holds the number of times it has acquired it before it can be acquired by any other thread. This is true in several languages including Java. Our predictive techniques can be extended to languages that do not permit re-entrant locks. Instead of counting how many times a lock is being acquired so that it is considered available for other threads after it has been released that many times, the analyses can be modified to consider a lock available after it has been released once by the thread that held it and to consider a thread blocked if it tries to re-acquire a lock it already has. We also assume that locks are acquired and released in a nested fashion, i.e., if a thread acquires lock l_2 after acquiring lock l_1, then it has to release l_2 before releasing l_1. In other words, a thread always releases the last lock that it had acquired. Though the predictive testing techniques that we have developed assume the nesting of lock acquires and releases, they can be extended to arbitrary locking patterns. We made the assumption to simplify the program analyses in our predictive testing techniques, and also because the assumption holds true for Java, a language that we have focused on in the implementations of our techniques.

Threads can also communicate by explicitly sending notifications to other threads. A thread can block itself and wait on a monitor [57] for a condition to become true. When

another thread sets the condition to true, it can notify a thread (if any) that is waiting on a monitor for that condition, or it can notify all threads that are waiting for that condition. A condition or predicate and the queue of threads waiting for that condition to become true are represented by a condition variable. A thread can wait on a condition variable of a monitor by blocking and adding itself to the queue associated with the condition variable, and it can remove and release a thread or all threads from the queue to notify those threads.

3.1 Events

In our predictive testing analyses, we consider the following statements that are executed by a thread. Each statement has a label that distinguishes the statement from other statements. We sometimes drop the statement label when it is not needed in the context of our discussion. We call the execution of a statement as an event or an operation.

1. c: `Acquire`(l): acquire of the dynamic lock l. c is the label of the statement (same for below).
2. c: `Release`(l): release of lock l. l is the last lock that was acquired by the executing thread.
3. c: `Wait(`l`)`: waiting on the condition variable (monitor) l.
4. c: `Notify(`l`)`: notifying a thread (if any) that is waiting on the condition variable l.
5. c: `NotifyAll(`l`)`: notifying all threads waiting on the condition variable l.
6. c: `Start(`t`)`: starting of a fresh thread t, that is, a thread that has not yet been started.
7. c: `Join(`t`)`: waiting for thread t to finish executing.
8. c: `Call`(o, m): invocation of method m on object o.
9. c: `Return`(m): return from the method m.
10. c: $o =$ `new` (o', T): allocation of a dynamic object o of type T. The statement occurs in the body of a method m when m is invoked by object o'.
11. c: `Write(`o`)`: writing to (some field of) object o.

When designing a predictive testing technique, we have to figure out the right set of events we should keep track of to effectively and efficiently detect the kind of bugs that we want to find. The above statements sufficed for the predictive analyses that we developed, but we might need to keep track of additional statements when designing a different analysis for detecting a different class of errors. For example, for data races, one needs to keep

track of both reads and writes, and record the field information along with other relevant information for those operations.

We often denote an event by e, and denote the thread in which the event executes by $\texttt{thr}(e)$. In our analyses, we sometimes need to know the set of locks held by a thread when executing an event to determine if the event is properly synchronized against other events. We call the set of locks held by a thread as the lockset of the thread and denote the lockset of a thread when executing an event e as $\mathcal{LS}(e)$.

We define a happens-before relation between events that we use in our analyses. Given a sequence of events $\langle e_i \rangle$ (e_i represents the execution of a statement), we define a happens-before relation $\prec$ as follows. $\prec$ is the smallest relation satisfying the following conditions:

- If e_i and e_j are events in the same thread and e_i comes before e_j in the sequence $\langle e_i \rangle$, then $e_i \prec e_j$.
- If e_i is the execution of `Start(`t`)`, then for the first event e_j in thread t, $e_i \prec e_j$.
- If e_i is the execution of `Join(`t`)`, then for the last event e_j in thread t, $e_j \prec e_i$.
- If e_i is the execution of `Notify(`l`)` or `NotifyAll(`l`)`, then for each event e_j that is the execution of `Wait(`l`)` and that was blocked before the execution of e_i, $e_i \prec e_j$.
- If events e_i, e_j, and e_k are such that $e_i \prec e_j$ and $e_j \prec e_k$, then $e_i \prec e_k$.

The happens-before relation $\prec$ essentially captures the inter-thread dependencies that will be respected in any feasible execution of the program under test. For example, if in a program, a thread t_1 starts another thread t_2, then no event in t_2 can ever execute before the event in t_1 that starts t_2 in any feasible execution of the program. The temporal dependencies between the event that starts t_2 and the events in t_2 will be captured by the happens-before relation. For each event e of type `Start(`t`)` or `Join(`t`)` or `Notify(`l`)` or `NotifyAll(`l`)` or `Wait(`l`)`, we can consider it to be either sending a message to another thread, or receiving a message from another thread. For example, if e is `Start(`t`)`, then it sends a message to thread t, and the first event in thread t receives the message. Similarly, if e is `Join(`t`)`, then the last event in thread t sends a message, and e receives the message. If e is `Notify(`l`)` or `NotifyAll(`l`)`, then e sends a message, and the `Wait(`l`)` that gets unblocked receives the message. The happens-before relation relates a sender event to its corresponding receiver event.

The happens-before relation can be computed during program execution by maintaining a vector clock [37, 81, 86] with every thread. Each thread t_i maintains a vector clock indexed by thread IDs. t_i's vector clock entry for t_j indicates the last event in t_j that could have affected t_i. Let $\mathrm{V}(t)$ denote the vector clock of thread t. When thread t is created, $\mathrm{V}(t)$ is initialized to 1 for t and 0 for other threads, that is, $\mathrm{V}(t)(t) = 1$ and $\mathrm{V}(t)(t') = 0$ for all $t' \neq t$. We also associate a vector clock with each monitor l and denote it by $\mathrm{V}(l)$. $\mathrm{V}(l)$ is initialized to 0 for all threads. The vector clock for a monitor l keeps track of the vector

clocks of threads when they execute waits and notifies on l. When an event e in thread t is executed, the vector clocks of different threads and monitors are updated in the following manner after the execution of e.

1. If e is `Start`(t'), then
   a) $\mathrm{V}(t)(t) = \mathrm{V}(t)(t) + 1$
   b) $\mathrm{V}(t')(t_k) = \mathtt{max}(\mathrm{V}(t')(t_k), \mathrm{V}(t)(t_k))$ for all $t_k \neq t'$
2. If e is `Join`(t'), then
   a) $\mathrm{V}(t)(t_k) = \mathtt{max}(\mathrm{V}(t)(t_k), \mathrm{V}(t')(t_k))$ for all $t_k \neq t$
   b) $\mathrm{V}(t)(t) = \mathrm{V}(t)(t) + 1$
3. If e is `Wait`(l), then
   a) $\mathrm{V}(t)(t_k) = \mathtt{max}(\mathrm{V}(t)(t_k), \mathrm{V}(l)(t_k))$ for all $t_k \neq t$
   b) $\mathrm{V}(t)(t) = \mathrm{V}(t)(t) + 1$
4. If e is `Notify`(l) or `NotifyAll`(l), then
   a) $\mathrm{V}(t)(t) = \mathrm{V}(t)(t) + 1$
   b) $\mathrm{V}(l)(t_k) = \mathrm{V}(t)(t_k)$ for each thread t_k

We also associate a vector clock with the event e and denote it by $\mathrm{V}(e)$. $\mathrm{V}(e)$ is the vector clock of thread t after e has been executed in it. For two vector clocks V_1 and V_2, we say $\mathrm{V}_1 < \mathrm{V}_2$ iff $((\forall\, t_i\ \mathrm{V}_1(t_i) \leq \mathrm{V}_2(t_i)) \wedge (\exists\, t_j\ \mathrm{V}_1(t_j) < \mathrm{V}_2(t_j)))$.

Theorem 1 *$(e_i \prec e_j) \Leftrightarrow (V(e_i) < V(e_j))$*

Proof We first prove $(e_i \prec e_j) \Rightarrow (\mathrm{V}(e_i) < \mathrm{V}(e_j))$, and then $(\mathrm{V}(e_i) < \mathrm{V}(e_j)) \Rightarrow (e_i \prec e_j)$. Also, let V_i denote $\mathrm{V}(e_i)$ and V_j denote $\mathrm{V}(e_j)$.

Assume $e_i \prec e_j$. First consider the case where e_i and e_j execute in the same thread t. Thus, e_i executes before e_j. Since the elements in the vector clock of a thread never decrease during the course of an execution, $\forall\, t_k\ \mathrm{V}_i(t_k) \leq \mathrm{V}_j(t_k)$. Furthermore, since $\mathrm{V}(t)(t)$ increases by 1 for each event executed in t, $\mathrm{V}_i(t) < \mathrm{V}_j(t)$. Therefore, $\mathrm{V}_i < \mathrm{V}_j$. Now consider the case where e_i executes in thread t_i and e_j in thread t_j and $t_i \neq t_j$. We have the following four possibilities.

- e_i is the execution of `Start`(t_j) and e_j is the first event in t_j. Then, $\forall\, t \neq t_j\ (\mathrm{V}_i(t) \leq \mathrm{V}_j(t))$. Also, $\mathrm{V}_i(t_j) = 0$ as t_i has not received any communication from t_j and hence it does not have any knowledge of the events in t_j, but $\mathrm{V}_j(t_j) > 0$. Thus, $\mathrm{V}_i(t_j) < \mathrm{V}_j(t_j)$. This implies that $\mathrm{V}_i < \mathrm{V}_j$.

- e_j is the execution of `Join`(t_i) and e_i is the last event in t_i. Then, $\forall\ t \neq t_j$ $V_i(t) \leq V_j(t)$. Since $V_j(t_j)$ is incremented by 1 after execution of e_j, $V_i(t_j) < V_j(t_j)$. Thus, $V_i < V_j$.

- e_i is the execution of `Notify`(l) or `NotifyAll`(l) for monitor l and e_j is the `Wait`(l) that was unblocked by the execution of e_i. Since e_i is the last notification on l before the wait e_j was woken up, $V(l) = V_i$ when computing V_j. Thus, $\forall\ t \neq t_j$ $V_i(t) \leq V_j(t)$ and $V_i(t_j) < V_j(t_j)$. Therefore, $V_i < V_j$.

- There exists an event e_k such that $e_i \prec e_k$ and $e_k \prec e_j$. Using the previous three cases and the associativity of $<$ for vector clocks, we can prove that $V_i < V_j$.

We now prove $(V_i < V_j) \Rightarrow (e_i \prec e_j)$. This is equivalent to proving $\neg\ (e_i \prec e_j) \Rightarrow \neg\ (V_i < V_j)$. Since $\neg\ (e_i \prec e_j)$, there is no chain of messages between the threads starting from e_i and ending at e_j. Since the vector clock of a thread updates the entry for another thread only when it receives a message from the other thread, $V_j(t_i) < V_i(t_i)$. Thus, $\neg\ (V_i < V_j)$.

3.2 Multithreaded execution state

We use the following definitions regarding the state of threads and the multithreaded execution.

- `Enabled`(s) denotes the set of threads that are enabled in the state s. A thread is disabled if it is waiting to acquire a lock already held by some other thread or waiting to be notified by another thread or waiting for another thread to finish executing.

- `Alive`(s) denotes the set of threads whose executions have not terminated in the state s. A state s is in a *stall state* if the set of enabled threads in s (i.e. `Enabled`(s)) is empty and the set of threads that are alive (i.e. `Alive`(s)) is non-empty.

- `Execute`(s, t) returns the multithreaded execution state after executing the next statement of the thread t in the state s.

We use the definitions and formalizations presented in this chapter to explain our predictive techniques in the subsequent chapters.

Chapter 4

Resource Deadlocks

A deadlock in a multithreaded program is an unintended condition in which one or more threads block forever waiting for a synchronization event that will never happen. Deadlocks are a common problem in real-world multithreaded programs. For instance, 6,500/198,000 ($\sim 3\%$) of the bug reports in the bug database at `http://bugs.sun.com` for Sun's Java products involve the keyword "deadlock" [67]. There are a few reasons for the existence of deadlocks in multithreaded programs. First, software systems are often written by many programmers; therefore, it becomes difficult to follow a synchronization discipline that could avoid deadlock. Second, programmers often introduce deadlocks when they fix race conditions by adding new locks. Third, software systems can allow incorporation of third-party software (e.g. plugins); third-party software may not follow the synchronization discipline followed by the parent software and this sometimes results in deadlock bugs [69].

Figure 4.1 illustrates a simple deadlock. There are two threads, `T1` and `T2`, with each trying to synchronize data by acquiring some locks. The first thread, `T1`, acquires the lock `L1`. The second thread, `T2`, acquires the lock `L2`. Thread `T1` then tries to acquire the lock `L2` but cannot since thread `T2` is still holding that lock. When thread `T2` tries to acquire the lock `L1`, it cannot as it is still held by thread `T1`. Thus, `T1` and `T2` get blocked with each waiting for the other to release a lock. We have a deadlock as a result. Deadlocks of this kind are called as *resource deadlocks* because the blocked threads are waiting for other blocked threads to release resources (or locks). Note in the figure that we have a cycle with the edges between `L1` and `L2`. Most of the tools for finding resource deadlocks [53, 13, 6, 112, 34] look for the presence of such cycles in appropriate graphs or data structures that encode the locking information.

4.1 Predictive testing for resource deadlocks

Most of the previous work [52, 53, 13, 6, 112, 34, 79, 42, 8, 107, 84] for resource deadlocks report deadlocks by finding cycles in the locking patterns across different threads. Lock-graph is a data structure that is used to express succinctly the locking patterns of threads.

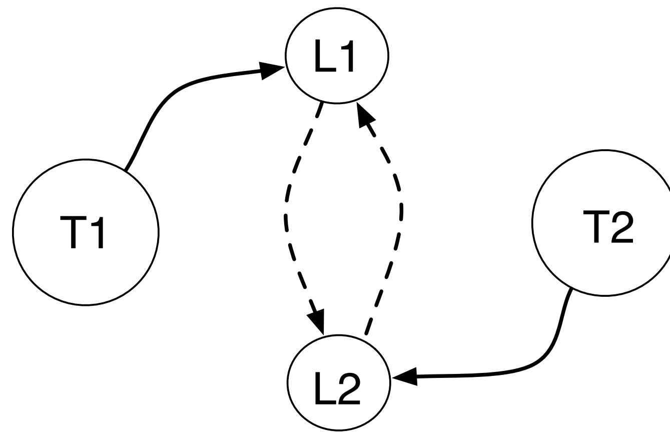


Figure 4.1: Example of a resource deadlock

It has a node for each lock that is acquired by a thread, and has a labeled edge from one lock to the other if a thread acquires the latter lock while holding the former lock in some program state. Previous predictive dynamic techniques [53, 13, 6] compute the lock-graph dynamically during program execution, and find cycles in it to predict deadlocks that could have occurred in a different thread interleaving. The algorithm in their predictive analyses is called as the Goodlock algorithm. The analyses use the basic Goodlock algorithm [53] or generalized variations of the basic algorithm [13, 6].

We built an improved version of the generalized Goodlock algorithm [13, 6] to predict resource deadlocks [67]. We improve over Goodlock in the following two ways. First, we add context information to a computed potential deadlock cycle. This information helps to identify the program locations where the deadlock could happen and also to statically identify the lock and thread objects involved in the deadlock cycle. This aids in debugging of deadlocks, and as we would later see (Chapter 7) that it also helps in confirming the deadlocks automatically. Second, we simplify the generalized Goodlock algorithm by avoiding the construction of a lock graph. Unlike existing Goodlock algorithms, we do not perform a depth-first search of the lock-graph, but compute transitive closure of the lock dependency relation (defined below in Section 4.3). As such it uses more memory, but has better runtime complexity. We call our version of the Goodlock algorithm as the iGoodlock algorithm with the "i" standing for "informative". We describe the iGoodlock algorithm in the following sections.

4.2 Overview

In this section, we illustrate how iGoodlock works with the help of the Java program in Figure 4.2. The program defines a `MyThread` class that has two locks `l1` and `l2` and a

```
 1 class MyThread extends Thread {
 2   Object l1, l2;
 3   boolean flag;
 4   MyThread(Object l1, Object l2, boolean b){
 5     this.l1 = l1; this.l2 = l2; this.flag = b;
 6   }
 7
 8   public void run() {
 9     if (flag) { // some long running methods
10       f1();
11       f2();
12       f3();
13       f4();
14     }
15     synchronized(l1) {
16       synchronized(l2) {
17       }
18     }
19   }
20
21   public static void main (String[] args) {
22     Object o1 = new Object();
23     Object o2 = new Object();
24     (new MyThread(o1, o2, true)).start();
25     (new MyThread(o2, o1, false)).start();
26   }
27 }
```

Figure 4.2: Simple example of a deadlock

boolean `flag`. The `run` method of `MyThread` invokes a number of long running methods `f1, f2, f3, f4` if `flag` is true and then it acquires locks `l1` and `l2` in order. The body of `run` shows a common pattern, where a thread runs several statements and then acquires several locks in a nested way. The `main` method creates two lock objects `o1` and `o2`. It also creates two threads (i.e. instances of `MyThread`). In the first instance `l1` and `l2` are set to `o1` and `o2`, respectively, and `flag` is set to true. Therefore, a call to `start` on this instance will create a new thread which will first execute several long running methods and then acquire `o1` and `o2` in order. A call to `start` on the second instance of `MyThread` will create a new thread which will acquire `o2` and `o1` in order.

The example has a deadlock because the locks `o1` and `o2` are acquired in different orders by the two threads. However, this deadlock will rarely occur during normal testing because the second thread will acquire `o2` and `o1` immediately after start, whereas the first thread will acquire `o1` and `o2` after executing the four long running methods. iGoodlock will report this deadlock as a potential one by observing a single execution that does not deadlock.

iGoodlock keeps track of each lock acquire and release during execution. For each lock

acquired by a thread, it records some information regarding that lock acquire that later helps iGoodlock to predict deadlocks. For example, when the second thread (say `T2`) acquires the lock on `o2` at line number 15, iGoodlock will record this event as the tuple (`T2`, [], `o2`, [15]). The first entry in the tuple denotes the thread that acquired the lock, the second entry represents the set of locks that the thread already held when acquiring the lock, the third entry is the lock, and the last entry is the set of source locations where the thread has acquired the locks that it currently holds. Similarly, iGoodlock generates the tuples (`T2`, [`o2`], `o1`, [15, 16]), (`T1`, [], `o1`, [15]) and (`T1`, [`o1`], `o2`, [15, 16]]) to record the other three lock acquires that it observes. All of these four tuples together form the lock dependency relation (Section 4.3). After the program execution finishes, iGoodlock uses the computed lock dependency relation to find potential deadlock cycles. Essentially, it tries to find cycles among the tuples. For example, the tuples (`T2`, [`o2`], `o1`, [15, 16]) and (`T1`, [`o1`], `o2`, [15, 16]]) form a cycle since it is possible in an execution that thread `T2` is waiting to acquire lock `o1` that is held by thread `T1`, and `T1` is waiting to acquire the lock on `o2` that is held by `T2`. This cycle represents the deadlock in the program that we had seen before.

4.3 iGoodlock

We first introduce some formal definitions before we describe the algorithm.

Given a multithreaded execution σ, let L_σ be the set of lock objects that were held by any thread in the execution and T_σ be the set of threads executed in the execution. Let $\mathcal{C}$ be the set of all statement labels in the multithreaded program. In iGoodlock, we need to keep track of the execution of the `Acquire` and `Release` statements (Chapter 3). Thus, $\mathcal{C}$ includes the labels of only `Acquire` and `Release` statements. Moreover, we ignore an `Acquire` statement when it is a re-acquire of a previously held lock, and we also ignore a `Release` statement when it does not release a lock completely. This is implemented by associating a usage counter with a lock which is incremented whenever a thread acquires or re-acquires the lock and decremented whenever a thread releases the lock. Execution of `Acquire`(l) by thread t is considered whenever t acquires or re-acquires the lock l and the usage counter associated with l is incremented from 0 to 1. Similarly, execution of `Release`(l) is considered when the usage counter goes from 1 to 0.

We next define *the lock dependency relation* of a multithreaded execution as follows.

Definition 2 *Given an execution σ, a* lock dependency relation *D_σ of σ is a subset of $T_\sigma \times 2^{L_\sigma} \times L_\sigma \times \mathcal{C}^*$ such that $(t, L, l, C) \in D_\sigma$ iff in the execution σ, in some state, thread t acquires lock l while holding the locks in the set L, and C is the sequence of labels of `Acquire` statements that were executed by t to acquire the locks in $L \cup \{l\}$.*

For the example in Figure 4.2, {(`T2`, [], `o2`, [15]), (`T2`, [`o2`], `o1`, [15, 16]), (`T1`, [], `o1`, [15]), (`T1`, [`o1`], `o2`, [15, 16])} forms the lock dependency relation.

Definition 3 *Given a lock dependency relation D,* a lock dependency chain $\tau = \langle(t_1, L_1, l_1, C_1), \ldots, (t_m, L_m, l_m, C_m)\rangle$ *is a sequence in D^* such that the following properties hold.*

1. *for all distinct $i, j \in [1, m]$, $t_i \neq t_j$, i.e. the threads $t_1, t_2, \ldots, t_m$ are all distinct objects,*
2. *for all distinct $i, j \in [1, m]$, $l_i \neq l_j$, i.e. the lock objects $l_1, l_2, \ldots, l_m$ are distinct,*
3. *for all $i \in [1, m-1]$, $l_i \in L_{i+1}$, i.e. each thread could potentially wait to acquire a lock that is held by the next thread,*
4. *for all distinct $i, j \in [1, m]$, $L_i \cap L_j = \emptyset$, i.e., each thread t_i should be able to acquire the locks in L_i without waiting.*

Consider the example in Figure 4.2 again. ⟨ (`T2`, [`o2`], `o1`, [15, 16]), (`T1`, [`o1`], `o2`, [15, 16]) ⟩ and ⟨(`T1`, [`o1`], `o2`, [15, 16]), (`T2`, [`o2`], `o1`, [15, 16])⟩ are two lock dependency chains in that example.

Definition 4 *A lock dependency chain*
$\tau = \langle(t_1, L_1, l_1, C_1), \ldots, (t_m, L_m, l_m, C_m)\rangle$
is a potential deadlock cycle *if $l_m \in L_1$.*

Both the dependency chains that we saw earlier are potential deadlock cycles for the example in Figure 4.2.

Note that the definition of a potential deadlock cycle never uses any of the C_i's in D_σ to compute a potential deadlock cycle. Each C_i of a potential deadlock cycle provides us with information about program locations where the locks involved in the cycle were acquired. This is useful for debugging and as we would later see (Chapter 7) is also used to determine the program locations at which to pause threads when confirming a potential deadlock.

We next describe iGoodlock. Specifically, we describe how we compute the lock dependency relation during a multithreaded execution and how we compute all potential deadlock cycles given a lock dependency relation.

4.3.1 Computing the lock dependency relation of a multithreaded execution

In order to compute the lock dependency relation during a multithreaded execution, we maintain the following three data structures:

- `LockSet` that maps each thread to a stack of locks held by the thread. This is essentially the lockset ($\mathcal{LS}$ in Chapter 3) of each thread, but maintained as a stack and not a set.
- `Context` that maps each thread to a stack of the labels of statements where the thread acquired the currently held locks

- D is the lock dependence relation

We update the above three data structures during a multithreaded execution as follows:

- Initialization
  - for all t, both `LockSet`$[t]$ and `Context`$[t]$ map to an empty stack
  - D is an empty set
- If thread t executes the statement c: `Acquire`(l)
  - push c to `Context`$[t]$
  - add (t, `LockSet`$[t]$, l, `Context`$[t]$) to D
  - push l to `LockSet`$[t]$
- If thread t executes the statement c: `Release`(l)
  - pop from `Context`$[t]$
  - pop from `LockSet`$[t]$

At the end of the execution, we output D as the lock dependency relation of the execution. Note that we use stacks for `LockSet` and `Context` because we assume that locks are acquired and released in a nested fashion (Chapter 3). We would have to use appropriate data structures for other (or arbitrary) locking patterns. The runtime complexity of computing the lock dependency relation is O(S) where S is the number of synchronization operations (lock acquires and releases) since the data structures are updated only for lock acquires and releases and each update is O(1). The memory complexity of computing the relation is O(S^2) since the lockset or context saved for a lock acquire is bounded by O(S). In Figure 4.2, when thread `T2` acquires the lock on `o1` at line number 16, `LockSet`[`T2`] is {`o2`}, and `Context`[`T2`] is [15, 16]. Thus, we add (`T2`, {`o2`}, `o1`, [15, 16]) to the lock dependency relation for that lock acquire.

4.3.2 Computing *potential* deadlock cycles iteratively

Let D^k denote the set of all lock dependency chains of D that has length k. Therefore, $D^1 = D$. iGoodlock computes potential deadlock cycles by iteratively computing $D^2, D^3, D^4, \ldots$ and finding deadlock cycles in those sets. The iterative algorithm for computing potential deadlock cycles is described in Algorithm 1. iGoodlock uses abstractions to identify the threads and locks involved in a deadlock when reporting the deadlock. For example, an abstraction for a thread or a lock object could be the source location where that object was created. In a later chapter (Chapter 7), we show more precise abstractions that can distinguish better between distinct objects. The method `abs` in Algorithm 1 computes the abstractions of objects in a deadlock cycle. The runtime complexity of computing the deadlock cycles is O(2^S), where S is the number of synchronization operations (lock acquires

Algorithm 1 `iGoodlock`(D)

```
1: INPUTS: lock dependency relation D
2: i ⇐ 1
3: D^i ⇐ D
4: while D^i ≠ ∅ do
5:     for each (t, L, l, C) ∈ D and each τ in D^i do
6:         if ⟨τ, (t, L, l, C)⟩ is a dependency chain by Definition 3 then
7:             if ⟨τ, (t, L, l, C)⟩ is a potential deadlock cycle by Definition 4 then
8:                 report abs(⟨τ, (t, L, l, C)⟩) as a potential deadlock cycle
9:             else
10:                add ⟨τ, (t, L, l, C)⟩ to D^{i+1}
11:            end if
12:        end if
13:    end for
14:    i ⇐ i + 1
15: end while
```

and releases). A lock dependency chain can have a maximum length of T where T is the number of threads, and thus, we can compute the dependency chains in T steps with each step i computing chains of length i. Since there are O(S) tuples in the lock dependency relation, the number of chains computed in step i is $O(S^i)$, and therefore the runtime complexity of each step is also $O(S^i)$. Hence, the overall runtime complexity is $O(S^1 + S^2 + \ldots + S^T)$ $= O(2^S)$. The memory complexity of computing deadlock cycles is $O(2^S TS)$ since there are $O(2^S)$ chains that are computed and each chain is bounded by O(TS). For the example in Figure 4.2, D^1 is {(`T2`, [], `o2`, [15]), (`T2`, [`o2`], `o1`, [15, 16]), (`T1`, [], `o1`, [15]), (`T1`, [`o1`], `o2`, [15, 16])}, and D^2 is $\emptyset$ since both ⟨(`T2`, [`o2`], `o1`, [15, 16]), (`T1`, [`o1`], `o2`, [15, 16])⟩ and ⟨ (`T1`, [`o1`], `o2`, [15, 16]), (`T2`, [`o2`], `o1`, [15, 16])⟩ are reported as potential deadlock cycles.

Note that in `iGoodlock`(D) we do not add a lock dependency chain to D^{i+1} if it is a deadlock cycle. This ensures that we do not report complex deadlock cycles, i.e. deadlock cycles that can be decomposed into simpler cycles.

4.3.3 Avoiding duplicate deadlock cycles

In Algorithm 1, a deadlock cycle of length k gets reported k times. For example, if
$\langle (t_1, L_1, l_1, C_1), (t_2, L_2, l_2, C_2), \ldots, (t_m, L_m, l_m, C_m) \rangle$
is reported as a deadlock cycle, then
$\langle (t_2, L_2, l_2, C_2), \ldots, (t_m, L_m, l_m, C_m), (t_1, L_1, l_1, C_1) \rangle$
is also reported as a cycle. In order to avoid such duplicates, we put another constraint in Definition 3: the unique id of thread t_1 must be less than the unique id of threads $t_2, \ldots, t_m$. With this constraint, we report only one potential deadlock cycle ⟨ (`T1`, [`o1`], `o2`, [15, 16]), (`T2`, [`o2`], `o1`, [15, 16])⟩ and not two as before.

Thus, we have seen how resource deadlocks can be predicted using a deadlock-free execution. In the next chapter, we present a technique that not only predicts resource deadlocks but also other classes of deadlocks that involve waits and signals.

Chapter 5

Generalized Deadlocks

Resource deadlocks (Chapter 4) are not the only reason why threads can get blocked forever during execution. There are other reasons for a multithreaded execution to go into a deadlock. For example, incorrect use of condition variables (i.e. wait-notify synchronization) as well as unintended interaction between locks and condition variables can also result in deadlocks. Such deadlocks are called as *communication deadlocks*, and are no less widespread or insidious than resource deadlocks. Yet, in the extensive literature on deadlock detection only little work (e.g. [5, 60]) addresses communication deadlocks. Thus, we set out to design a predictive testing technique that can predict communication deadlocks.

The iGoodlock algorithm (Chapter 4) that we developed for resource deadlocks, and most of the previous work on resource deadlock detection [6, 13, 52, 53, 67, 16, 34, 60, 80, 84, 107, 112] find deadlocks by checking for the violation of a simple idiom, namely that there is no cycle in the lock-graph of a program. Thus, as a first attempt, we tried to build a tool that can predict communication deadlocks by checking for the violation of an idiom analogous to that for resource deadlocks. However, after studying a large number of communication deadlocks in real-world programs, we realized that there is no single idiom that programmers follow when writing code using condition variables. Therefore, any such testing technique based on checking idioms would give many false positives and false negatives. The study also suggested that finding communication deadlocks is a hard problem as a communication deadlock can be non-trivially dependent on aspects of the underlying synchronization logic that vary from program to program.

In the end, we adopted a completely different approach [66] that we describe in the subsequent sections. Instead of checking conformance to a particular idiom, we create a simple multithreaded program, called a *trace program*, by observing an execution of the original program, and model check the trace program to discover potential deadlocks. We can find a broad class of deadlocks or *generalized deadlocks* with the trace program that include both communication deadlocks and resource deadlocks and deadlocks involving both locks and condition variables. The trace program for a given multithreaded execution creates an explicit thread for each dynamic thread created by the execution. The code for each thread in the trace program consists of the sequence of lock acquires and releases, wait

```
  // F1                          // F2
synch (l) {                    synch (l) {
  while (!b)                     <change in value
    l.wait();                    of b that could
  <do something that             make b true>
  requires b = true>             l.notifyAll();
}                              }
```

Figure 5.1: Recommended condition variable usage. We use synch to abbreviate synchronized.

and notify calls, and thread start and join calls by the corresponding thread in the original execution. However, this is not enough: the state of the *synchronization predicates* associated with condition variables must also be tracked and checked. In the trace programs, these synchronization predicates are represented as boolean variables with explicit assignments where the original program performed an assignment that caused the predicate's value to change, and explicit checks before uses of waits and notifies. We call our predictive testing technique as CheckMate, and describe it in detail in the following sections in the context of Java that we have focused on in our implementation of the technique.

5.1 Rationale

In this section, we explain the rationale behind our approach after describing the recommended usage pattern for condition variables.

Recommended Usage

Figure 5.1 shows the recommended pattern for using condition variables in Java [1]. The condition variable in the figure is associated with a synchronization predicate b (e.g. "FIFO X is non-empty"). Any code like $F1$ that requires b to be true (e.g. "dequeuing an element from FIFO X") must hold the condition variable's lock l, and repeatedly wait (l.wait()) on the condition variable until b is true. Any code $F2$ that might make b true (e.g. "enqueue an element in FIFO X") must make these modifications while holding the condition variable's lock l, and must notify all threads that might be waiting for b to become true (l.notifyAll()). After a waiting thread wakes up, it should again check to see if b indeed holds for two reasons. First, the notifying thread ($F2$) may notify when b is not necessarily true. Second, some other thread may have invalidated b before the woken thread was able to acquire lock l.

[1]Strictly speaking, Java uses monitors that combine a lock and a condition variable. We use lock and condition variable to clarify which aspect of a monitor we are referring to; this also makes clearer how CheckMate would apply to languages with separate locks and condition variables.

Our Initial Effort

As a first attempt, we devised a predictive testing technique that predicted communication deadlocks by checking to see if all condition variables used in observed thread interleavings followed the recommended usage pattern shown in Figure 5.1. Consider the real-world code fragment in Figure 5.2. Any of its interleavings violates two aspects of that pattern: first, thread 1 uses an `if` instead of a `while` to check predicate b, and secondly, neither thread accesses b in the same synchronized block as the one containing l`.wait()` or l`.notifyAll()`. The analysis we devised thus reports a possible deadlock in this code fragment regardless of the interleaving it observes. Indeed, the shown interleaving of this code fragment exhibits a deadlock. In this interleaving, thread 1 first finds that boolean b is false. Thread 2 then sets b to true, notifies all threads in the wait set of l (i.e. the threads that are waiting on l), and releases l. The wait set of l, however, is empty; in particular, thread 1 is not (yet) waiting on l. Finally, thread 1 resumes and waits on l, assuming incorrectly that b is still false, and blocks forever as thread 2—the only thread that could have notified it—has already sent a notification and terminated. This is a classic kind of communication deadlock called a *missed notification* in Java.

Limitations of Pattern Enforcement

We found pattern-based enforcement to be of limited value, for two reasons. First, programmers often optimize the recommended pattern based on their knowledge about the code. For instance, if the synchronization predicate b is always true when a thread is woken up, then the thread may not need to check b again, i.e. the `while` in $F1$ can be an `if`. Or, if a number of threads are woken up by a notifying thread, but the first thread that acquires lock l always falsifies the predicate b, then waking up the other threads is pointless. In this case, the notifying thread ($F2$) can use `notify` to wake a single thread. A real-world example violating the pattern is found in `lucene` (version 2.3.0), a text search engine library by Apache: in some cases, a thread may change the value of a synchronization predicate in one synchronized block and invoke `notifyAll()` in another synchronized block. Although the notification happens in a different synchronized block, it always follows the change in value of the predicate, and hence there is no deadlock because of this invariant which the recommended usage pattern does not capture.

Second, code that respects the pattern can still deadlock because of interactions between other locks and condition variables. Consider the code in Figure 5.3 (based on a real example): the locks follow the cycle-free lock-graph idiom for avoiding resource deadlocks, and the condition variables follow the recommended usage pattern in Figure 5.1 for avoiding communication deadlocks, yet their combined use causes a deadlock, exhibited by the shown interleaving. The second thread is the only thread that can wake the first thread up, but it gets blocked when trying to acquire lock `l1` as it is still held by the first thread. This code fragment can occur because a library uses the condition variable involving lock `l2`, and an application calls into the library while holding its own lock `l1`. In fact, this pattern occurs

```
// Thread 1                          // Thread 2
  if (!b)
                                       b = true;
                                       synch (l)
                                          l.notifyAll();
  synch (l)
     l.wait();
```

Figure 5.2: Deadlock due to missed notification.

```
// Thread 1                          // Thread 2
  synch (l1)
     synch (l2)
        while (!b)
           l2.wait();
                                       synch (l1)
                                          synch (l2)
                                             l2.notifyAll();
```

Figure 5.3: Deadlock involving both locks and condition variables.

frequently enough in practice that FindBugs, a popular static bug-finding tool for Java that checks common bug patterns, reports that calls to `wait()` with two locks held may cause a deadlock [60].

In summary, we could not find an idiom for accurately predicting all deadlocks by observing interleavings that did not exhibit them. This motivated us to devise an analysis that uses a model checker to explore all possible interleavings. Model checking is difficult to scale to large programs. We chose to strike a trade-off between scalability, completeness, and soundness by model checking a *trace program* obtained from a single execution of the given program. In doing so, we sacrifice both completeness and soundness, but our analysis scales to large programs. This is not only because the trace program is generated from a single finite execution of the given program, but also because it only records operations that we deem are relevant to finding the above kinds of deadlocks. Not only does our analysis find both deadlocks discussed above, it does not report a false deadlock for the correct usage of notification in `lucene` described above.

5.2 Overview

In this section, we illustrate how CheckMate works using the example Java program in Figure 5.4. Class `MyBuffer` is intended to implement a thread-safe bounded buffer that allows a producer thread to add elements to the buffer and a consumer thread to remove elements from it. List `buf` represents the buffer, `cursize` denotes the current number of

elements in the buffer, and `maxsize` denotes the maximum number of elements allowed in the buffer at any instant. Ignore the underlined field `condition` and all operations on it for now. The program uses two condition variables to synchronize the producer and consumer threads, one for the predicate that the buffer is full, checked using method `isFull()`, and the other for the predicate that the buffer is empty, checked using method `isEmpty()`.

A producer thread adds elements to the buffer using the `put()` method. If the buffer is full, it waits until it gets notified by a consumer thread. After adding an element to the buffer, it notifies any consumer thread that may be waiting for elements to be available in the buffer. Likewise, a consumer thread removes elements from the buffer using the `get()` method. If the buffer is empty, it waits until it gets notified by a producer thread. After removing an element from the buffer, if the buffer was full, it notifies any producer thread that may be waiting for space to be available in the buffer. Finally, the `resize()` method allows changing the maximum number of elements allowed in the buffer.

The `main()` method creates a `MyBuffer` object `bf` with a `maxsize` of 1. It also creates and spawns three threads that execute in parallel: a producer thread `p` that adds two integer elements to `bf`, a consumer thread `c` that removes an element from `bf`, and a third thread `r` that resizes `bf` to have a `maxsize` of 10.

Suppose we execute the program, and the three threads spawned by the main thread interleave as shown in Figure 5.5. In this interleaving, thread `p` first puts integer 0 into `bf`. Since the `maxsize` of `bf` is 1, `bf` is now full. But before `p` puts another integer into `bf`, thread `r` changes the `maxsize` of `bf` to 10. Thus, `bf` is not full any more. Thread `p` then puts integer 1 into `bf`. Finally, thread `c` removes integer 0 from `bf`. Note that neither of the two `wait()`'s in the program is executed in this interleaving. However, there is another interleaving of threads `p`, `r`, and `c` that deadlocks. This interleaving is shown in Figure 5.6. Thread `p` puts integer 0 into `bf`. Since the `maxsize` of `bf` is 1, `bf` gets full. When `p` tries to put another integer into `bf`, it executes the `wait()` in the `put()` method and blocks. Thread `r` then increases the `maxsize` of `bf`, and thus, `bf` is not full any more. Thread `c` then removes integer 0 from `bf`. Since `bf` is not full any more (as thread `r` grew its capacity), it does not notify thread `p`. Thus, `p` blocks forever.

Our analysis can predict the deadlock from the interleaving in Figure 5.5, although that interleaving does not exhibit the deadlock, and does not even execute any `wait()`. For this purpose, our analysis records three kinds of information during the execution of that interleaving. First, it records synchronization events that occur during the execution, like lock acquires and releases, calls to `wait()` and `notify()`, and calls to `start()` and `join()` threads. Secondly, it records changes to the value of any predicate associated with a condition variable during the execution. Since Java has no explicit synchronization predicates associated with condition variables, our analysis requires the user to explicitly identify each such predicate by defining an instance of class `ConditionAnnotation` (shown in Figure 5.8). In our example in Figure 5.4, there are two condition variables in class `MyBuffer`, one for predicate `isFull()`, and the other for predicate `isEmpty()`. We manually annotate the `MyBuffer` class with the underlined field `condition` defined on lines 4-8 to identify predicate `isFull()`. This field holds a `ConditionAnnotation` object that defines a method

```
1 public class MyBuffer {
2   private List buf = new ArrayList();
3   private int cursize = 0, maxsize;
4   private ConditionAnnotation condition =
5      new ConditionAnnotation(this) {
6         public boolean isConditionTrue() {
7           return ((MyBuffer) o).isFull();
8         }};
9   public MyBuffer(int max) {
10     maxsize = max;
11  }
12  public synch void put(Object elem) {
13     condition.waitBegin(this);
14     while (isFull()) wait();
15     condition.waitEnd();
16     buf.add(elem); cursize++; notify();
17  }
18  public Object get() {
19     Object elem;
20     synch (this) {
21        while (isEmpty()) wait();
22        elem = buf.remove(0);
23     }
24     synch (this) {
25        condition.notifyBegin(this);
26        if (isFull()) {
27           cursize--; notify();
28        } else {
29           condition.notifyEnd(); cursize--;
30        }
31     }
32     return elem;
33  }
34  public synch void resize(int max) { maxsize = max; }
35  public synch boolean isFull() { return (cursize >= maxsize); }
36  public synch boolean isEmpty() { return (cursize ==  0); }
37  public static void main(String[] args) {
38     final MyBuffer bf = new MyBuffer(1);
39     Thread p = (new Thread() {
40        public void run() {
41           for (int i = 0; i < 2; i++) bf.put(new Integer(i));
42        }}).start();
43     Thread r = (new Thread() { public void run() { bf.resize(10);}}).
          start();
44     Thread c = (new Thread() { public void run() { bf.get();}}).start();
45  }
46 }
```

Figure 5.4: An example with a communication deadlock.

```
// Thread p          // Thread r          // Thread c
  bf.put(0)
                     bf.resize(10)
  bf.put(1)
                                            bf.get()
```

Figure 5.5: Non-deadlocking interleaving for example in Figure 5.4.

```
// Thread p          // Thread r          // Thread c
  bf.put(0)
  bf.put(1)
                     bf.resize(10)
                                            bf.get()
```

Figure 5.6: Deadlocked interleaving for example in Figure 5.4.

`isConditionTrue()` that can determine in any program state whether that predicate is true. Our analysis uses this method to determine if each write in the observed execution changes the value of predicate `isFull()`. We provide a similar annotation (not shown for brevity) for predicate `isEmpty()`. Note that our annotations are very simple to add if we know the implicit synchronization predicates associated with condition variables.

Thirdly, our analysis also records each `wait()` and `notify()` event that did not occur during the observed execution because the condition under which it would have occurred was false in that execution. Our analysis again relies on manual annotations for this purpose, this time in the form of calls to pre-defined methods `waitBegin()`, `waitEnd()`, `notifyBegin()`, and `notifyEnd()` on the `ConditionAnnotation` object corresponding to the predicate associated with the condition. The annotations on lines 13 and 15 denote that the execution of `wait()` in the `put()` method depends on the value of predicate `isFull()`. During execution, even if this predicate is false, these annotations enable our analysis to record that had it been true, the `wait()` would have executed. Likewise, the annotations on lines 25 and 29 denote that the execution of `notify()` in the `get()` method depends on the value of predicate `isFull()`. We provide similar annotations (not shown for brevity) to indicate that the execution of `wait()` in the `get()` method depends on predicate `isEmpty()`.

Our analysis generates the Java program shown in Figure 5.7, which we call a *trace program*, by observing the execution of the interleaving in Figure 5.5 and with the help of the above annotations. Note that the trace program has excluded all the complex control structure (e.g. the for loop and method calls) and memory updates (e.g. changes in `cursize` and `buf`) and has retained the necessary synchronization operations that happened during the execution. This simple trace program without the complicated program logic of the original program is much more efficient to model check.

In the trace program, we have used descriptive identifier names and comments to help relate it to the original program. Such comments and identifier names help our analysis

```
1 public class TraceProgram {
2   static Object bf = new Object();
3   static boolean isFull;
4   static Thread main = new Thread() {
5      public void run() {
6         isFull = false;
7         p.start();
8         r.start();
9         c.start();
10     }
11  };
12  static Thread p = new Thread() {
13     public void run() {
14        synch (bf) {    // enter bf.put(0)
15           if (isFull) {
16             synch (bf) { bf.wait(); }
17           }
18           isFull = true;
19           bf.notify();
20        }               // leave bf.put(0)
21        synch (bf) {    // enter bf.put(1)
22           if (isFull) {
23              synch (bf) { bf.wait(); }
24           }
25           bf.notify();
26        }               // leave bf.put(1)
27     }
28  };
29  static Thread r = new Thread() {
30     public void run() {
31        synch (bf) { // enter bf.resize(10)
32           isFull = false;
33       }             // leave bf.resize(10)
34     }
35  };
36  static Thread c = new Thread() {
37     public void run() {
38        synch (bf) {    // enter bf.get()
39           if (isFull) {
40              synch (bf) { bf.notify(); }
41           }
42        }               // leave bf.get()
43     }
44  };
45  public static void main(String[] args) {
46     main.start();
47  }
48 }
```

Figure 5.7: Trace program generated by observing the execution of the interleaving in Figure 5.5 of the example in Figure 5.4.

to map any error trace in the trace program to the original program, which could be used for debugging. The fact that our analysis can generate an informative error trace in the original program is a key advantage of our technique over other predictive dynamic analysis techniques. In the trace program, `bf` denotes the instance of `MyBuffer` created during the observed execution. Note that we make `bf` of type `Object`, instead of type `MyBuffer`, because we do not need to worry about the program logic in the trace program. `isFull` denotes predicate `bf.isFull()` upon which the `wait()` in the `put()` method and `notify()` in the `get()` method are control-dependent. The main thread `main` initializes `isFull` to false, and starts threads `p`, `r`, and `c` as in the observed execution. Note that although the `wait()` in the `put()` method is not executed in either of the two calls to `bf.put()` by thread `p` in that execution, the `run()` method of thread `p` in the trace program records that this `wait()` would have been executed in either call had `isFull` been true. Also, `isFull` is set to true on line 18 since the buffer becomes full after thread `p` puts the first integer 0 into it. Thread `r` is the thread that resizes the buffer and increases its `maxsize`. The `run()` method of thread `r` sets `isFull` to false on line 32 since the buffer is no longer full after its `maxsize` has been increased. Finally, although the `notify()` in the `get()` method is not executed in the call to `bf.get()` by thread `c` in the observed execution, the `run()` method of thread `c` in the trace program records that the `notify()` would have been executed had `isFull` been true. Thus, the trace program captures all synchronization events in the observed execution, any writes in that execution that change the value of any annotated predicate associated with a condition variable, as well as any annotated `wait()`'s and `notify()`'s that did not occur in that execution but could have occurred in a different execution. All other operations in the observed execution of the original program are not deemed relevant to finding deadlocks.

There exists an interleaving of the threads in this trace program that corresponds to the interleaving in Figure 5.6 that exhibits the deadlock. In this interleaving of the trace program, `p` executes its `run()` method till the `wait()` on line 23, where it gets blocked. Then, `r` completely executes its `run()` method and exits. Thereafter, `c` executes its `run()` method, but does not notify `p` because `isFull` is false. Thus, `p` blocks forever waiting to be notified by `c`. CheckMate uses an off-the-shelf model checker to explore all possible interleavings of the trace program and check if any of them deadlocks. In the process of model checking, it encounters this interleaving, and thus finds the deadlock in the original program.

5.3 Predictive Testing for Communication Deadlocks

In this section, we present our predictive deadlock detection algorithm. We first describe the annotations our algorithm requires. We then discuss how to generate the trace program, and how to model check the trace program to report possible deadlocks in the original program.

```
1 public abstract class ConditionAnnotation{
2   protected static int counter = 0;
3   protected Object o;
4   protected int condId;
5   protected boolean curVal;
6   public ConditionAnnotation(Object o1) {
7      o = o1; condId = counter++;
8      associateWithObject(o1); initCond();
9   }
10  public abstract boolean isConditionTrue();
11  public void waitBegin(Object lock) {
12     int lockId = getUniqueObjId(lock);
13     boolean val = isConditionTrue();
14     addLine("if (c"+condId+") {");
15     if (!val)
16       addLine("synchronized (l"+lockId+")"+"{l"+lockId+".wait();}");
17  }
18  public void waitEnd() { addLine("}"); }
19  public void notifyBegin(Object lock) {
20     int lockId = getUniqueObjId(lock);
21     boolean val = isConditionTrue();
22    addLine("if (c"+condId+") {");
23     if (!val)
24       addLine("synchronized (l"+lockId+")"+"{l"+lockId+".notify();}");
25  }
26  public void notifyEnd() { addLine("}"); }
27  public void logChange() {
28     boolean newVal = isConditionTrue();
29     if (newVal != curVal) {
30       addLine("c"+condId+"="+newVal+";");
31       curVal = newVal;
32     }
33  }
34  private void associateWithObject(Object o) {
35     /** associate this instance of ConditionAnnotation with the object o in a map **/
36     ...
37  }
38  private void initCond() {
39     curVal = isConditionTrue();
40     addLine("c"+condId+"="+curVal+";");
41  }
42  private void addLine(String line) {
43     AddLinesToTraceProgram.addLine(line);
44  }
45 }
```

Figure 5.8: Definition of class `ConditionAnnotation`.

```
1 public class AddLinesToTraceProgram {
2   protected static Map/*<Int,List<String>>
3      */ thrToLines = new TreeMap();
4   public static int getUniqueObjId(Object o) {
5       ... // return unique integer ID for object o
6   }
7   public static void addLine(String line)
8   {
9       int tId = getUniqueObjId(
10          Thread.currentThread());
11      /** append line to list mapped to tId in thrToLines **/
12      ...
13  }
14 }
```

Figure 5.9: Definition of class `AddLinesToTraceProgram` that is used by class `ConditionAnnotation` (Figure 5.8) and Algorithm 2.

5.3.1 Condition Annotations

Our algorithm requires users to annotate the predicate associated with each condition variable in a Java program using class `ConditionAnnotation`. This class is shown in Figure 5.8. For brevity, we have not shown the synchronization required to make the `ConditionAnnotation` class thread-safe in the figure. Here, we only explain the annotation and how the user provides it. We later show how our algorithm uses it to generate the trace program.

For the predicate associated with a given condition variable, the user subclasses `ConditionAnnotation` to implement its only abstract method `isConditionTrue()`. This method should evaluate to true if and only if the predicate evaluates to true. The predicate usually depends on the state of an object in the program, that is, the predicate's value can be computed by accessing fields or calling methods of that object. Field o of `ConditionAnnotation` denotes that object. In practice, we have multiple such fields to accommodate cases where the predicate depends on the state of multiple objects. We can also handle predicates that depend on static variables. The user thus implements method `isConditionTrue()` by accessing the appropriate fields, or calling the appropriate methods of o. Then, whenever the object on whose state the predicate depends is created in the program, the user also creates an instance of the defined subclass of `ConditionAnnotation`, and associates it with that object by passing it to `ConditionAnnotation`'s constructor. Finally, the user calls pre-defined methods `waitBegin()`, `waitEnd()`, `notifyBegin()`, and `notifyEnd()` on the created instance of `ConditionAnnotation` before and after any calls to `wait()`, `notify()`, and `notifyAll()` that are control-dependent on the predicate.

The example from Figure 5.4 uses two condition variables, one for the predicate that the buffer is full, and another for the predicate that it is empty. The underlined field `condition`

Algorithm 2 TRACEPROGRAMGENERATOR(s_0)

```
1: s ⇐ s0
2: while Enabled(s) ≠ ∅ do
3:     t ⇐ a random thread in Enabled(s)
4:     stmt ⇐ next statement to be executed by t
5:     s ⇐ Execute(s, t)
6:     if stmt = Acquire(l) then
7:         lId ⇐ getUniqueObjId(l)
8:         addLine( "synchronized (l" + lId + ") {" )
9:     else if stmt = Release(l) then
10:        addLine( "}" )
11:    else if stmt = Wait(l) then
12:        lId ⇐ getUniqueObjId(l)
13:        addLine( "l" + lId + ".wait();" )
14:    else if stmt = Notify(l) then
15:        lId ⇐ getUniqueObjId(l)
16:        addLine( "l" + lId + ".notify();" )
17:    else if stmt = NotifyAll(l) then
18:        lId ⇐ getUniqueObjId(l)
19:        addLine( "l" + lId + ".notifyAll();" )
20:    else if stmt = Start(t) then
21:        tId ⇐ getUniqueObjId(t)
22:        addLine( "t" + tId + ".start();" )
23:    else if stmt = Join(t) then
24:        tId ⇐ getUniqueObjId(t)
25:        addLine( "t" + tId + ".join();" )
26:    else if stmt = Write(o) || Stmt = Call(o, m) then
27:        for each ConditionAnnotation c associated with o do
28:            c.logChange()
29:        end for
30:    end if
31: end while
32: if Alive(s) ≠ ∅ then print 'System Stall!' endif
33: CreateTraceProgram(AddLinesToTraceProgram.thrToLines)
```

defined on lines 4-8 in the figure, along with the methods invoked on it on lines 13, 15, 25, and 29 to specify control-dependent calls to `wait()` and `notify()`, constitutes the annotation for the predicate that the buffer is full. Method `isConditionTrue()`, in this case, simply calls method `isFull()` of class `MyBuffer`. A similar annotation (not shown) is provided for the predicate that the buffer is empty.

Algorithm 3 `CreateTraceProgram(thrToLines)`

```
1: lockIds ⇐ set of lock identifiers in thrToLines
2: predIds ⇐ set of synchronization predicate identifiers in thrToLines
3: thrIds ⇐ set of thread identifiers in thrToLines
4: print "public class TraceProgram {"
5: for all lId such that lId is in lockIds do
6:     print "  static Object l" + lId + " = new Object();"
7: end for
8: for all pId such that pId is in predIds do
9:     print "  static boolean p" + pId + ";"
10: end for
11: for all tId such that tId is in thrIds do
12:     print "  static Thread t" + tId + " = new Thread() {"
13:     print "    public void run() {"
14:     for all s such that s is in thrToLines[tId] do
15:         print s
16:     end for
17:     print "    } };"
18: end for
19: print "  public static void main(String[ ] args) {"
20: for all tId: tId + ".start();" not in any list in thrToLines do
21:     print tId + ".start();"
22: end for
23: print "  } }"
```

5.3.2 Stage I: Generating the Trace Program

The first stage of our algorithm generates a trace program by observing an execution of the given program with annotations on synchronization predicates. Algorithm 2 explains how the trace program is generated. It populates global map `thrToLines` in class `AddLinesToTraceProgram` (Figure 5.9) while observing the execution. Map `thrToLines` maps each thread in the observed execution to a list of strings. Each string is a statement or a part of a statement, and the whole list is a legal block of statements that constitutes the body of the corresponding thread in the trace program. Whenever a synchronization statement is executed by a thread in the observed execution, the algorithm calls method `addLine()` in class `AddLinesToTraceProgram` to add the string it generates to the list mapped to that thread in `thrToLines`. The generated string depends on the kind of synchronization statement that was executed.

If a lock acquire statement (i.e. statement of type `Acquire` as in Chapter 3) is executed, the algorithm begins a `synchronized` statement for the involved lock object. The lock object is uniquely identified in the trace program using method `getUniqueObjId()` in class `AddLinesToTraceProgram` which provides a unique integer for each object created in the observed execution. If a lock release statement is executed, the algorithm closes the last

`synchronized` statement that it had started for the thread. We had earlier stated our assumption that locks are acquired and released in a nested fashion (Chapter 3). Thus, a lock release statement always releases the lock that was most recently acquired by the thread.

If a `wait()` statement is executed, the algorithm generates a corresponding `wait()` statement. Similarly, when a `notify()`, `notifyAll()`, `start()`, or `join()` statement is executed, a corresponding statement is generated.

If a statement writing to (some field of) object o is executed, or a method is called on an object o, then the algorithm finds all `ConditionAnnotation` objects that are associated with object o. After the write or the method call, the state of o may have changed, and hence the predicates associated with those `ConditionAnnotation` objects may have also changed. The trace program needs to track changes to the values of predicates associated with condition variables. For this purpose, the algorithm calls method `logChange()` in class `ConditionAnnotation` to evaluate the predicate and check if its value has indeed changed, and if so, generates a statement writing the new value to the variable associated with the predicate.

When calls to the pre-defined methods `waitBegin()`, `waitEnd()`, `notifyBegin()`, and `notifyEnd()` (Figure 5.8) that have been added as annotations execute, statements are generated for the trace program that capture the control-dependence of the execution of the `wait()` or `notify()` or `notifyAll()` on the synchronization predicate that has been annotated. These statements are also added using the method `addLine()` in class `AddLinesToTraceProgram`.

After observing the complete execution, the algorithm creates a legal Java program (trace program) by calling method `createTraceProgram()` defined in Algorithm 3. It creates an object for each lock object, a boolean variable for each predicate, and a thread object for each thread in the observed execution. For each created thread, it prints a `run()` method containing the list of strings generated for the corresponding thread in map `thrToLines`. Finally, the algorithm prints the `main()` method, and starts all those threads in it which are not started by any other thread [2]. The trace program for the example in Figure 5.4 is shown in Figure 5.7.

The algorithm does a couple of optimizations before it prints the body for each thread. Firstly, it does not print any `synchronized` statement that involves a lock that is local to the thread. Thread-local locks cannot be involved in a deadlock, and hence, it removes statements that use them. Secondly, for any block of statements that consists only of `synchronized` statements, it does the following optimization. It finds the different nestings of lock acquires within the block. Instead of printing all `synchronized` statements present in the block, the algorithm prints one block of nested `synchronized` statements for each nesting of lock acquires. This removes a lot of redundancy in `synchronized` statements because of loops in the original program. If the original program has a loop, then the same pattern of lock acquires can be repeated a number of times. After the optimization, the

[2] normally just the main thread

algorithm prints only one block of nested lock acquires for the pattern, and does not repeat the pattern as many times as it was observed during program execution.

5.3.3 Stage II : Model Checking the Trace Program

The second stage of our algorithm uses an off-the-shelf model checker to explore all possible thread interleavings of the trace program and check if any of them deadlocks. When we generate the trace program, for each line in it, we also record the source location of the statement in the original program that led to its generation. This helps in mapping a deadlocking interleaving in the trace program (i.e. a counterexample) back to an interleaving in the original program. A deadlock in the trace program may or may not imply a deadlock in the original program. The counterexample provided by the model checker assists in determining whether a deadlock reported by our algorithm is real or false. Multiple counterexamples may denote the same deadlock. We group together counterexamples in which the same set of statements (either lock acquires or calls to `wait()`) is blocked and report each such group as a different possible deadlock. The deadlock for the example in Figure 5.4 is shown in Figure 5.6.

5.4 CheckMate and iGoodlock

Note that CheckMate can not only predict communication deadlocks but also resource deadlocks. The trace program captures the locking and wait-notify synchronization observed during execution, and therefore can predict a broad class of deadlocks that involve locks or condition variables or both. CheckMate can thus find all the resource deadlocks that can be predicted by iGoodlock that we saw in the previous chapter. But, iGoodlock has the advantage of being more efficient since it focuses only on resource deadlocks. Furthermore, iGoodlock is optimized to have a better runtime complexity than the basic iGoodlock algorithm. As shown in Section 4.3, iGoodlock has a runtime complexity of $O(2^S)$, where S is the number of synchronization operations (lock acquires and releases), but CheckMate has a runtime complexity of O(S!). In CheckMate, we explore all interleavings of the synchronization operations, and thus, its runtime complexity if O(S!). Therefore, the runtime complexity of CheckMate is more than that of iGoodlock. Our experiments (Chapter 9) also validate that iGoodlock is faster than CheckMate when finding resource deadlocks. However, the memory complexity of iGoodlock as shown in Section 4.3 ($O(2^S TS)$) is more than that of CheckMate (O(S)). The trace program generated by CheckMate has a memory complexity of O(S) since a constant number of statements are generated in the trace program for each synchronization operation observed. Thus, the memory complexity of CheckMate is O(S). A tester interested in finding resource deadlocks has to keep the trade-offs in mind when deciding on using iGoodlock or CheckMate.

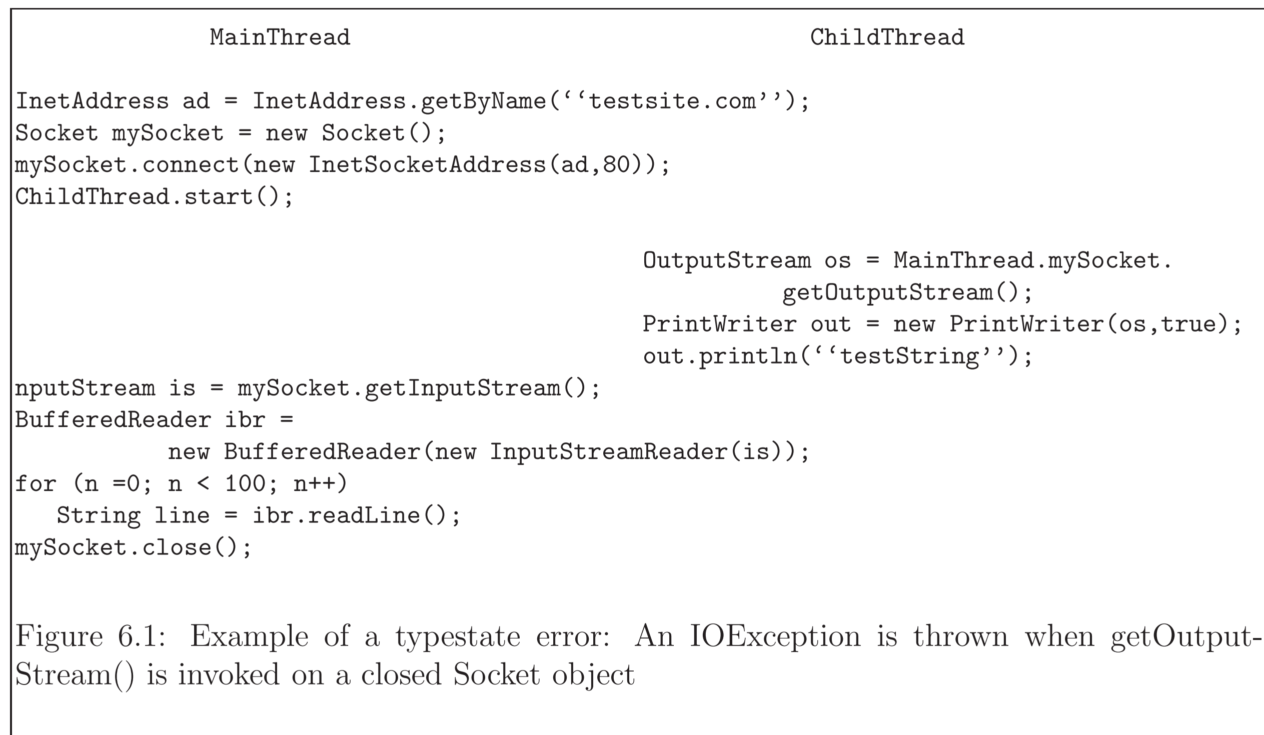
We have now seen predictive techniques for finding deadlocks. In the next chapter, we present a predictive technique for finding a different class of concurrency errors, typestate errors.

Chapter 6

Typestate Errors

Typestate [104] properties are high-level properties that specify the correct usage protocols of libraries and application programming interfaces (APIs). For example, a typestate property on an `InputStream` object in Java is that one cannot read from an `InputStream` object after it has been closed. The typestate property of an object type can be conveniently represented by a finite state automaton. A state in the automaton represents a typestate. At any state in an execution, an object can be exactly in one typestate, with the typestate of the object being set to the initial state of the automaton when the object is being created. Each edge of the typestate is labeled by a method defined by the object type. The invocation of a method during an execution changes the typestate of an object according to its typestate automaton. If a typestate in the automaton has no edge corresponding to a method, then an invocation of the method during an execution on an object in the typestate leads to a violation of the typestate property. Dynamically checking a typestate property [9] of an object simply involves checking that the sequence of method calls made on the object is a sequence accepted by the object's typestate automaton.

Figure 6.1 shows an example of a typestate error. The example has two threads: `MainThread` and `ChildThread`. `MainThread` creates a new `Socket` object, connects it to an address, and then starts `ChildThread`. `MainThread` obtains an input stream for the socket, and reads from it a number of times. Finally, `MainThread` closes the socket. `ChildThread` obtains an output stream for the socket, and writes a string to it. The example program is buggy and can throw an exception. Such an exception is thrown if `MainThread` is executed to completion before `ChildThread` executes its first statement. This is because at the completion of its execution, `MainThread` closes `mySocket` and then `ChildThread` calls `getOutputStream` on the closed socket. Such an execution violates the typestate property that the `getOutputStream` method of a `Socket` object cannot be called after calling the method `close` on the same object. However, in a normal execution it is very unlikely that `ChildThread` will be called after the completion of the execution of `MainThread`. This is because the execution of `MainThread` will take a long time due to the presence of the loop and a fair thread scheduler will schedule `ChildThread` long before `MainThread` completes its execution. Nevertheless, the exception can happen under some schedule and the bug in the



```
             MainThread                                        ChildThread

InetAddress ad = InetAddress.getByName(‘‘testsite.com’’);
Socket mySocket = new Socket();
mySocket.connect(new InetSocketAddress(ad,80));
ChildThread.start();

                                             OutputStream os = MainThread.mySocket.
                                                       getOutputStream();
                                             PrintWriter out = new PrintWriter(os,true);
                                             out.println(‘‘testString’’);
nputStream is = mySocket.getInputStream();
BufferedReader ibr =
           new BufferedReader(new InputStreamReader(is));
for (n =0; n < 100; n++)
   String line = ibr.readLine();
mySocket.close();
```

Figure 6.1: Example of a typestate error: An IOException is thrown when getOutputStream() is invoked on a closed Socket object

.

program should be fixed.

In our attempt to build a predictive testing technique for finding typestate errors, we had to overcome three problems [68]. First, checking typestate property for each object type is expensive and time-consuming. Second, coming up with the valid typestate property for each object type requires a lot of manual effort. Third, checking typestate property efficiently against all “nearby” thread interleavings could be expensive. We solve these problems by combining three techniques in three stages. In the first stage, we perform object race detection [108] to identify the object types whose methods could be concurrently invoked by multiple threads. Racing object types could only cause a typestate violation due to different interleavings in an execution; therefore, we only consider these object types in the next two stages. This helps us to significantly prune the object types whose typestate needs to be checked predictively. Second, we observe a successful execution, that is, an execution that does not throw an exception, and try to infer the likely typestate property of an object type by using an existing dynamic specification mining technique [7]. There are static methods to mine specifications [99] too, which cover all possible ways an object type can be used and not only the ways that were observed during an execution, but are usually not very scalable. Although our inferred typestate properties may not be accurate, they help to reduce the burden on specifications writer who can further help to refine the inferred specifications rather than trying to write them from scratch. Third, we efficiently check the inferred typestate properties by constructing an abstract model of an execution, called computation lattice. We call our predictive testing technique as PRETEX, and explain in

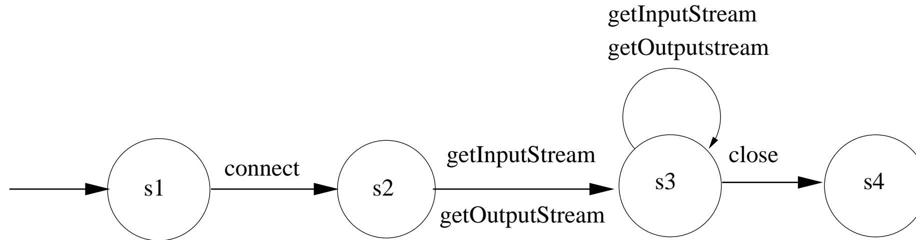


Figure 6.2: The likely typestate specification automaton learnt for Socket objects

in detail in the subsequent sections.

6.1 Overview

We explain how PRETEX works using the example in Figure 6.1. A naïve way to find the error in the program would be to execute the program many times with the hope that the thread scheduler will create the buggy thread interleaving in some execution. PRETEX can discover this typestate bug by just looking at a single successful execution of the program. We next explain how PRETEX predicts the occurrence of the bug by looking at a successful execution (or exception-free execution) where `ChildThread` is executed before `MainThread` calls the method `getInputStream`. The interleaving is shown in Figure 6.1.

PRETEX works in three stages. In the first stage, it computes the types of the objects whose method calls could potentially race with each other. Only the object types that could potentially race are considered for typestate checking in the next two stages. This is because the objects that could potentially race are likely to violate a typestate property due to lack of synchronization. Note that the first stage is only meant for optimization. In the example program, the `Socket` object, `mySocket`, is in race, since the method invocations, `getInputStream`() in `MainThread`, and `getOutputStream`() in `ChildThread`, can occur in either order.

Our second stage is the typestate specification mining stage. We use the typestate properties mined in this stage to predict if they could have been violated in some execution that was not observed but that could have occurred. In this stage, we infer the likely typestate property of each object type by observing a successful execution. Specifically, for each type obtained from the previous stage, we obtain the sequence of method calls invoked on each object of that type. We pass these sequences of method calls to an off-the-shelf machine learning procedure [92] to learn an automaton that contains all the observed sequences. Such an automaton represents the likely typestate property of the object type, i.e. the valid sequence of method calls on that object type.

For example, the sequence of calls on `mySocket`, the only `Socket` object, is `connect`, `getOutputStream`, `getInputStream`, `close` in a successful run where `ChildThread` termi-

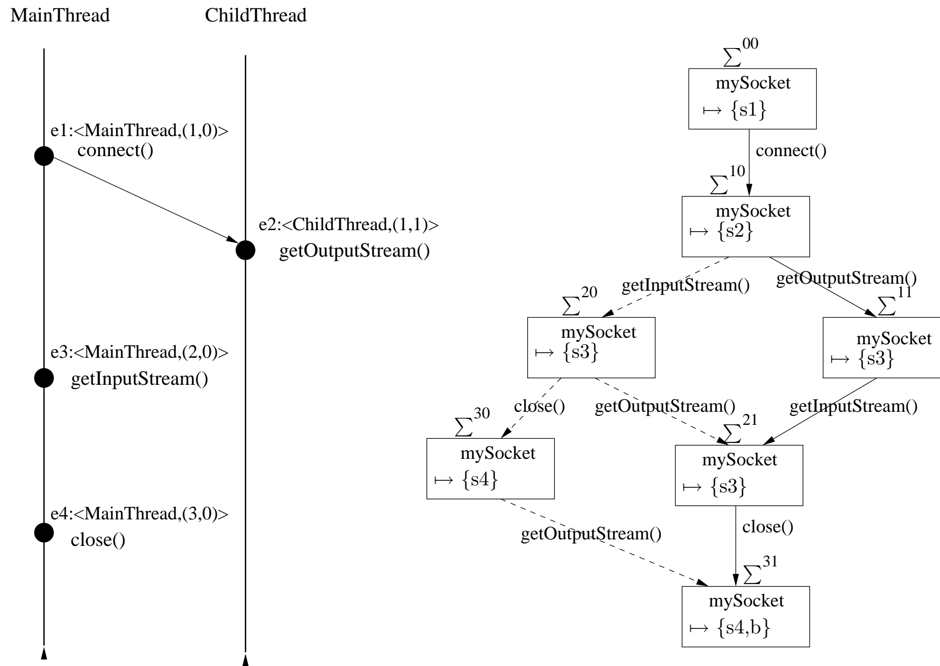


Figure 6.3: The multithreaded computation lattice for Figure 6.1

nates before `MainThread`. From this sequence, we can infer the likely typestate property that `getInputStream` or `getOutputStream`, cannot be called on a `Socket` object after `close` has been invoked on it. Typically, in real-world programs, we have a number of objects for each type in race, and hence, the automaton that describes the union of the sequences observed for these objects gets close to the correct typestate specification automaton for that type. Figure 6.2 gives a typestate specification automaton that can be inferred for `Socket` objects from our example.

The third stage is the predictive typestate checking stage. Once we have a likely typestate property for each type and a successful execution, we predictively check each typestate property against the successful execution. That is, by looking at the causal dependence among the various events in the successful execution, we compute other thread interleavings that can be obtained from the successful execution by reordering independent events. We then check each typestate property against the computed thread interleavings. In order to compute the thread interleavings "near" the successful execution, we compute an abstract model from the successful execution, called the *multithreaded computation lattice*. Each path in such a computation lattice denotes an execution that could potentially happen if we change the thread interleaving slightly. We then check the typestate property against all paths in this lattice using a dynamic programming algorithm.

Figure 6.3 gives the lattice for the observed execution in our example. The left-hand side diagram in the figure depicts the partial order observed during program execution. Each event is a pair of the thread name in which it occurred, and the vector clock of the thread

when it occurred. The state b in Σ^{31} in the computation lattice is the bad state in the typestate specification automaton for `Socket`. Any transition that is not possible in the specification automaton leads to the bad state b. The solid lines in the lattice describe the observed execution from which we have computed the entire lattice, whereas the dotted lines trace the executions that could have been observed. The program state Σ^{30} in the lattice shows that the typestate property of the `Socket` type can be violated. This is because the socket is already closed in this state, when the method `getOutputStream` is invoked on it. Note that we do not observe this state in the actual execution, but we predicted this erroneous state by analyzing the computation lattice.

6.2 Predictive Testing for Typestate Errors

We describe the different stages of PRETEX in this section. The first stage executes the program and finds the types of objects that are involved in race in those executions. The second stage executes the program again, once for each type that was identified to be in race in the previous stage, and obtains the sequence of method calls for each object of that type. PRETEX then constructs a typestate specification automaton for each type of object in race from the method sequences. The third and final stage predictively checks the inferred typestate specifications against a multithreaded execution.

6.2.1 Definitions

We describe the notion of a multithreaded computation lattice that we use in PRETEX. This lattice is computed from the execution of a multithreaded program as in [10, 97, 95, 98]. As we have seen in Chapter 3, a multithreaded program execution consists of a finite number of threads, and the execution can be viewed as a sequence of events where an event is the execution of a statement by a thread. Let us denote the k-th event that occurred in the i-th thread t_i by ${e_i}^k$. Then the program state after events ${e_1}^{k_1}$,${e_2}^{k_2}$, $\ldots$, ${e_n}^{k_n}$ is denoted by $\Sigma^{k_1 k_2 \ldots k_n}$. For example, in Section 6.1, the program state after the execution of `connect` by `MainThread` is denoted by Σ^{10}.

A state $\Sigma^{k_1 k_2 \ldots k_n}$ is called consistent [98] if and only if for any $1 \leq i \leq n$ and any $l_i \leq k_i$, $l_j \leq k_j$ holds for any $1 \leq j \leq n$ and any l_j such that ${e_j}^{l_j} \prec {e_i}^{l_i}$, where $\prec$ is the happens-before relation between events (Section 3.1). In other words, a consistent state is one which can be formed by an interleaving of events that respects the happens-before relation. For example, Σ^{10} is a consistent state in Section 6.1 that arises after the execution of `connect` by `MainThread`, but Σ^{01} is not a consistent state as `ChildThread` cannot execute its first event `getOutputStream` before `MainThread` executes `connect` and starts the `ChildThread`.

Let $\Sigma^{00 \ldots 0}$ be the initial program state. A feasible interleaving of events e_1, e_2, $\ldots$, e_m generates a sequence of program states Σ^{K_0}, Σ^{K_1}, $\ldots$, Σ^{K_m} for which the following two conditions hold. Each Σ^{K_r} is consistent, and for any two consecutive states, Σ^{K_r} and $\Sigma^{K_{r+1}}$, K_r and K_{r+1} differ in exactly one index by one. If the index in which the two differ is i,

then the i-th element of K_{r+1} is larger by one than the i-th element of K_r. A sequence of states Σ^{K_0}, Σ^{K_1}, ..., Σ^{K_m} thus identifies an interleaving of events or threads, or a run of the program. We say that Σ leads to Σ', written as $\Sigma \rightsquigarrow \Sigma'$ if there is a run in which Σ and Σ' are consecutive states. The set of all program states together with the partial order $\rightsquigarrow$ forms a lattice. For a state $\Sigma^{k_1 k_2 \ldots k_n}$, we call $k_1 + k_2 + \ldots + k_n$ as its level in the computation lattice. Consider the example in Section 6.1 again. Figure 6.3 shows the computation lattice for the example. An arrow in the figure from Σ to Σ' denotes $\Sigma \rightsquigarrow \Sigma'$.

6.2.2 Object race detection

This stage finds the types of objects whose method calls by different threads are in race. Our algorithm to detect the objects that are in race is a combination of the dynamic race detection techniques proposed in [108, 86]. Specifically, at runtime, we check the following condition for each pair of events (e_i, e_j).

$$e_i = \texttt{Call}(o_i, m_i) \wedge e_j = \texttt{Call}(o_j, m_j) \wedge (\texttt{thr}(e_i) \neq \texttt{thr}(e_j)) \wedge (o_i = o_j)$$
$$\wedge(\mathcal{LS}(e_i) \cap \mathcal{LS}(e_j) = \emptyset) \wedge \neg(e_i \prec e_j) \wedge \neg(e_j \prec e_i).$$

The definitions of `thr`, $\mathcal{LS}$, and happens-before relation ($\prec$) used in the above condition are provided in Chapter 3. The above condition essentially states that two events are in race if they are events on different threads, they are due to method calls on the same object, and the two events are not related by the happens-before relation (i.e. the two events are concurrent). If the above condition holds, we say that the object o_i could be in race, and we record its type. Objects which are in race are more likely to result in typestate errors. Therefore, we concentrate on building the typestate specification automaton for such objects in subsequent stages.

6.2.3 Inferring likely typestate specifications

Typestate [104] can be used to express the correct usage rules for many application programming interfaces (APIs). For example, one can use typestate to express that a `java.net.Socket` object cannot be read from after it has been closed. A typestate specification uses a finite state automaton (FSA) to encode the correct usage protocol. A state in the FSA is called a typestate, and an object is in one of these typestates at any point of time during program execution. The edges in the automaton are labeled with method names. When a method is invoked on an object, it follows that outgoing edge from its typestate which is labeled with the method name, and transitions to a new typestate (which might be the same as its old typestate). If no such edge exists in its current typestate, then we say that a typestate error has occurred. In this section, we briefly describe how we obtain the typestate specifications for object types that we identified to be in race in the previous stage.

For each type in race, we collect the sequence of method calls invoked on each object of that type. For example, if we find that objects of type `java.net.Socket` are in race, then for each object of type `java.net.Socket`, we record the sequence of method calls that was invoked on it during the execution of the program. Thus, we get as many sequences as the number of `Socket` objects that were used during the execution of the program. For each such set of sequences, we learn a finite state automaton (FSA) that accepts the sequences in the set, and rejects most of those outside the set. The automaton so learnt can be thought of as the typestate specification of that object type because it captures all the different ways objects of that type were used during program execution. Moreover, since the execution does not throw an exception, we can assume that the observed sequences of method calls are valid.

The FSA that we learn is a deterministic finite automaton (DFA), the edges of which are labeled with method names. We use an off-the-shelf PFSA (Probabilistic Finite State Automaton) learner [92]. The learner infers a PFSA that accepts the set of method call sequences presented to it, plus some more sequences that get added as it over-approximates. The PFSA learner first constructs a prefix tree or a trie from the set of sequences. A trie is a tree data structure with each path in the tree starting from the root down to a leaf representing a sequence. Each edge of the trie is labeled with a method name. The trie is initially empty with just the root node. Nodes and edges get added to the trie as we add sequences to it. To add a sequence, we first traverse as much of the sequence as we can in the trie starting from the root. We then add edges and nodes to create a path for the rest of the sequence. For example, to add the method sequence `connect`, `close` to the empty trie, we create an outgoing edge from the root labeled `connect`, and create a new node n to form the other end of the edge. We then add an outgoing edge from n labeled `close` and a new node m as the other end. Now, if we want to add the sequence `connect`, `getInputStream`, `close`, since we already have an edge labeled `connect` outgoing from the root, we traverse that edge and reach node n. There is no edge from n that is labeled `getOutputStream`. Thus, we create a new node k and label the edge from n to k with `getInputStream`. Finally, we create an outgoing edge from k labeled with `close`, and create a new node to form the other end of the edge.

Each edge of the trie is also labeled with a frequency that reflects how many times that edge was traversed while creating the trie. The trie can be seen as a FSA that accepts the set of sequences from which it was built. Since prefix trees are usually very large in size, the PFSA learner uses the sk-strings method [92] to merge nodes in the prefix trees. The sk-strings method is a variation of the k-tails method [14] for stochastic automata. It constructs a non-deterministic finite automaton (NFA) by successively merging those nodes of the trie which are sk-equivalent. Let Σ be the set of method names in the set of sequences, Q be the set of nodes in the trie, $\delta : Q \times \Sigma^* \to Q$ be the transition function of the trie (that is, the function that given a node and a sequence of method calls returns the node that would be reached if the sequence is traversed starting from the given node), and F_C be the final nodes (or leaves) of the trie. The set of k-strings of node q is then defined to be the set $\{z | z \in \Sigma^*, (|z| = k \wedge \delta(q, z) \in Q) \vee (|z| < k \wedge \delta(q, z) \in F_c)\}$ [92]. Each k-string

has a probability associated with it which is equal to the product of the probabilities of the edges traversed to form that string. The k-strings of a node are arranged in decreasing order of their probabilities. The top n strings, whose probabilities add up to $s\%$ or more, are retained and the rest discarded. Two nodes are said to be sk-equivalent if the sets of the top n strings of both are equal. The process of merging sk-equivalent nodes is repeated until no more nodes can be merged. The resulting PFSA accepts a superset of the method call sequences that was presented to it, due to the approximations done during merging. The final stage in the learning process converts the NFA into a DFA.

The DFA learned in this stage for each object type can be used as the likely typestate specification for the type. Note that we use this stage to reduce the burden on users so that they do not have to write tedious typestate specification for each type from scratch. However, they can take a look at the inferred typestate automata and refine them as required.

6.2.4 Predictive checking against typestate specifications

Algorithm 4 PRETEX: Level-by-level traversal of the computation lattice

```
Q ⇐ [ ]
while not end of computation do
    # get the next method invocation event observed
    e ⇐ NextEvent()
    add e to Q
    while constructLevel() do
        NOP
    end while
end while

boolean constructLevel() :
CurrLevel ⇐ [Σ^{00...0}]
NextLevel ⇐ [ ]
for each e ∈ Q do
    if Σ ∈ CurrLevel and isNextState(Σ, e) then
        NextLevel ⇐ NextLevel ⊎ createState(Σ, e)
        if isLevelComplete(NextLevel, e, Q) then
            Q ⇐ removeUselessEvents(CurrLevel, Q)
            CurrLevel ⇐ NextLevel
            return true
        end if
    end if
end for
return false
```

After we infer the likely typestate specification automata, we predictively check them against a multithreaded execution by generating a computation lattice for each automaton

based on method invocation events that are relevant to that automaton. Algorithm 4 illustrates the predictive checking algorithm. As the program under test executes, the method invocation events observed on objects whose typestates we want to check are queued into Q. `CurrLevel` denotes the current level of the computation lattice. It initially consists of just $\Sigma^{00\ldots0}$ which is the program state before any event has been observed. A program state is a mapping from objects to sets of states in the typestate specification automaton. Thus, the program state $\Sigma^{k_1 k_2 \ldots k_n}$ specifies the possible typestates that objects can be in after k_i ($1 \leq i \leq n$) events have been executed in the i-th thread in some order.

Algorithm 5 Check if it is feasible to execute an event e in the given program state Σ (**isNextState**(Σ, e))

```
i ⇐ thr(e)      # thread of event e
if (∀ j ≠ i : VC(Σ)[j] ≥ VC(e)[j] and VC(Σ)[i]+1 = VC(e)[i]) then
   return true
else
   return false
end if
```

Whenever a new event e is observed, the algorithm generates any new program state that can be obtained by executing e in a program state Σ already in `CurrLevel`. An event e can be executed in a program state Σ if the events that lead to Σ followed by e would form a feasible interleaving of events that respects the happens-before relation between events. Or, in other words, if e in Σ would result in a consistent state. For example, in Figure 6.1, let e denote the execution of the `getOutputStream` method on `mySocket` in `ChildThread`. If Σ is the initial state, then e in Σ would *not* result in a consistent state since the execution of `connect` in the `MainThread` should always happen before e in any feasible thread interleaving. But, if e is the execution of `connect`, then e in Σ would result in a consistent state. Algorithm 5 computes if event e in program state Σ results in a consistent state or not. It uses vector clocks (Section 3.1) to check if e in Σ would respect the happens-before relation or not. The vector clock of a program state Σ reflects the latest event that has occurred in each thread when the program state is reached. The vector clock of an event e is the vector clock of its thread when it occurred. In addition to updating the vector clocks as explained in Section 3.1, we also update the vector clocks when we observe a (method invocation) event e. We update the vector clock of the thread in which e executes (`thr`(e)) by 1 for that thread. `VC`(Σ) gives the vector clock associated with program state Σ, and `VC`(e) gives the vector clock associated with event e.

If the current event e can result in a consistent state with a state Σ in `CurrLevel`, then the new consistent state is generated and added to `NextLevel` that denotes the states in the next level of the lattice. Algorithm 6 illustrates how a new consistent state Σ' is computed. For the object o on which the method m is invoked in event e, the new typestates of o are computed using the typestate specification automaton for the object type of o. If the bad state b is one of the new typestates, then we report a typestate error for object o. Any

Algorithm 6 Return a new state after executing event e in state Σ (**createState**(Σ, e))

```
Σ' ⇐ new copy of Σ
j ⇐ thr(e)      # thread of event e
VC(Σ')[j] ⇐ VC(Σ)[j] + 1
Σ' ⇐ Σ'[obj(e) ⇐ ρ(Σ(obj(e)), methodId(e))]
if b ∈ ρ(Σ(obj(e)), methodId(e)) then
   print 'typestate error observed'
end if
return Σ'
```

transition that is not possible in the automaton is considered to lead to the bad state b. If the new state Σ' that is created has a vector clock equivalent to that of a state already in `NextLevel`, then we merge the two states. We merge the mappings of the corresponding objects in the two states. For an event e, `obj`(e) returns the object o and `methodId`(e) returns the method m. ρ takes the set of states `obj`(e) is mapped to in Σ, and returns the set of new states after applying the typestate automaton transition corresponding to `methodId`(e) to those states. For example, in Figure 6.1, when we observe `getInputStream` being executed, the current state is Σ^{11} and `mySocket` is in typestate `s3`. Using the typestate automaton in Figure 6.2, we find out that `mySocket` should remain in typestate `s3` after the invocation of `getOutputStream`. Thus, in the new state Σ^{21}, `mySocket` is mapped to typestate `s3`. We would generate the state Σ^{21} twice for the two different paths (or event orderings) in the lattice that lead to the state. In both the cases, we would find that `mySocket` would be in typestate `s3` in Σ^{21}. Had we found different typestates for the two cases, we would have merged the typestates into a set, and mapped `mySocket` to that set.

Algorithm 7 Check if a level in a lattice is complete (**isLevelComplete**(`NextLevel`, e, Q))

```
if size(NextLevel) ≥ w then
   return true
else if e is the last event in Q and size(Q) ≥ l then
   return true
else
   return false
end if
```

After new states have been generated using the current event e and the states in `CurrLevel`, the next level `NextLevel` is checked to see if it is complete or not. After a level is complete, we discard the previous level that was used to compute the new complete level, i.e. after `NextLevel` is complete, we discard `CurrLevel` and start computing the level after `NextLevel`. But, the problem with waiting till a level is complete is that the number of states in a level can be exponential in the number of that level. Thus, the number of states for the n-th level can be exponential in n. Instead of generating and storing an exponential number of states in a level, we employ a heuristic called the causality cone heuristic [98]

to cut down on the number of states. The heuristic is shown in Algorithm 7. Instead of generating all possible states in a level, the heuristic considers a level to be complete after w states in the level are generated, where w is a pre-determined parameter. However, a level may contain less than w states. The level construction algorithm would get stuck in that case. Also, one cannot determine if there are less than w states in a level unless one sees all the events in the complete computation. This is because the total number of threads is not known until the end of the execution. To avoid this, another parameter l is introduced. We consider the construction of a level to be complete if we have used all the events in the event queue Q for the construction of the states in that level and the length of queue is at least l, or if we have generated w states in that level. For example, in Figure 6.1, if we set l to 1 and w to 1, then after we have generated state Σ^{11}, we would consider the 2nd level to be over and would not later generate Σ^{20}.

After we are done with generating states for `NextLevel`, we discard the previous level `CurrLevel`, and use the states in `NextLevel` to generate states in the level after `NextLevel`. We also purge the event queue Q to discard events that can no longer generate consistent states when executed in any state in the recentmost level. The function `removeUselessEvents` in Algorithm 4 performs the purging. It creates a vector clock VC_{min}, each component of which is the minimum of the corresponding components of the vector clocks of the states in the recentmost level. All events in Q which have a vector clock less than or equal to VC_{min} are removed, because they cannot generate consistent states any more. For example, in Figure 6.1, after we are done with level 1, we can throw the `connect` invocation from Q since `connect` can no longer participate in generating new states later.

The algorithm to generate the lattice is very similar to the one presented in [98], except for the happens-before relation employed by them. The happens-before relation in [98] considers shared variable reads and writes, and lock acquires and releases, along with the synchronization events, `start()`, `join()`, `wait()`, `notify()` and `notifyAll()`. We do not consider shared variable reads and writes, and lock acquires and releases to avoid the overhead that would be incurred if we kept track of them. The happens-before relation (Section 3.1) that we employ is, thus, an over-approximation of the exact happens-before relation that exists between the events in a multithreaded execution, but it helps us in verifying more thread interleavings, some of which might be feasible, but not possible under the more conservative happens-before relation in [98].

In the next chapter, we show how we can try to automatically confirm the real errors in a set of predicted errors. A predicted error may or may not correspond to a real error in the program. Thus, if we can automatically confirm the real errors, then the developer does not have to go through the often tedious process of manually reasoning about each predicted error to see if it can occur in a real program execution or not.

Chapter 7

Confirming Predicted Errors

The predictive analyses that we saw in the previous chapters detect errors that could have occurred in a thread interleaving that did not show up during normal testing. These predicted errors are potential errors, that is, they might occur in a real interleaving of the program under consideration, or they might not occur in any feasible interleaving of the program. Thus, if we report all the predicted errors to the tester, then she would have to examine each error and investigate if it can occur in a real interleaving or not. The process of manually confirming errors can easily get tedious for real-world programs. Thus, after predicting errors, we try to automatically confirm which of the predicted errors are real errors. We do this by actively controlling the thread scheduler and by pausing threads at the right points during execution so that the execution is forced to go into a state that would likely exhibit a predicted error if it is a real error.

In order to identify the right preemption points during execution, the analysis that predicts the errors needs to also record some "extra" information for each predicted error. For example, for confirming a deadlock, we need the source locations of the lock acquires, and waits and notifies that are involved in the deadlock so that we can pause threads when they reach those locations to reproduce the deadlock. In this chapter, we describe a technique called DeadlockFuzzer that we designed to confirm resource deadlocks. Although DeadlockFuzzer confirms only resource deadlocks, it can be extended for other kinds of bugs. As we describe DeadlockFuzzer, we also explain how it can be adapted to confirm other classes of bugs.

7.1 Overview

Consider the example in Figure 4.2 that we saw earlier in Chapter 4. We reproduce the example here in Figure 7.1. Ignore the two extra statements that have been commented out. As we had seen earlier in Section 4.2, the example has a resource deadlock that is detected by iGoodlock and reported as the cycle ⟨(`T1`, [`o1`], `o2`, [15, 16]]), (`T2`, [`o2`], `o1`, [15, 16])⟩, with `T1` and `T2` denoting the first and the second thread respectively. To confirm whether

```
 1 class MyThread extends Thread {
 2   Object l1, l2;
 3   boolean flag;
 4   MyThread(Object l1, Object l2, boolean b){
 5     this.l1 = l1; this.l2 = l2; this.flag = b;
 6   }
 7
 8   public void run() {
 9     if (flag) { // some long running methods
10       f1();
11       f2();
12       f3();
13       f4();
14     }
15     synchronized(l1) {
16       synchronized(l2) {
17       }
18     }
19   }
20
21   public static void main (String[] args) {
22     Object o1 = new Object();
23     Object o2 = new Object();
24     // Object o3 = new Object();
25     (new MyThread(o1, o2, true)).start();
26     (new MyThread(o2, o1, false)).start();
27     // (new MyThread(o2, o3, false)).start();
28   }
29 }
```

Figure 7.1: Confirming a Resource Deadlock

the reported deadlock is a real deadlock or not, DeadlockFuzzer executes the program again. During execution, when the second thread `T2` tries to acquire the lock on `o1` at line number 16 after acquiring the lock on `o2` at line number 15, it pauses the thread. Note that the state of `T2` when it tries to acquire the lock on `o2` matches the state that is expressed in the tuple (`T2`, [`o2`], `o1`, [15, 16]) in the reported deadlock. Thus, DeadlockFuzzer essentially checks before each lock acquire to see if the state of the thread acquiring the lock matches the state expressed in any of the tuples in the reported deadlock. If it does, then it pauses the thread at that point. Similarly, DeadlockFuzzer pauses thread `T1` when it tries to acquire the lock on `o2` after having acquired the lock on `o1`. Since both the threads are now paused, DeadlockFuzzer releases one of them to let the execution proceed. If it releases `T1`, then `T1` gets blocked since `T2` holds the lock on `o2`. Similarly, if it releases `T2`, then `T2` gets blocked since `T1` has the lock on `o1`. There is a deadlock as a result. Thus, by pausing threads at the right points during execution, DeadlockFuzzer has confirmed

that the reported deadlock is a real deadlock.

For confirming other classes of bugs, the predictive analysis for those bugs should report information about the state of each thread when a particular bug occurs so that the bug can be confirmed by preempting threads when they reach their respective states. Each tuple in the deadlock cycle reported by iGoodlock approximates the state of a thread involved in the deadlock when the deadlock occurs. DeadlockFuzzer uses the thread states summarized in the reported deadlock to force all threads involved in the deadlock to reach their respective states simultaneously. For other kinds of bugs, the predictive analysis should provide appropriate approximation of the thread states when a bug occurs. The checker for those bugs can then preempt threads accordingly to confirm those bugs.

7.2 Confirming Resource Deadlocks

The iGoodlock algorithm described in Chapter 4 predicts resource deadlocks that can occur in thread interleavings that did not show up during normal testing. Let $\langle (t_1, L_1, l_1, C_1), (t_2, L_2, l_2, C_2), \ldots, (t_m, L_m, l_m, C_m) \rangle$ be a deadlock cycle computed by the algorithm. To recap what each element of a tuple in the cycle means: for $i \in [1, \ldots, m]$, t_i denotes a thread involved in the deadlock, L_i denotes the set of locks that the thread already holds when it acquires the lock on l_i, and C_i denotes the source locations where the thread has acquired all its locks. We call C_is as the contexts in which the locks should be acquired for the deadlock to occur.

To confirm a potential deadlock, DeadlockFuzzer needs to identify the locks and threads involved in the deadlock when it executes the program again. For this, iGoodlock associates a unique id to each thread and lock in the deadlock cycle that does not change across executions. The *address* of a thread or a lock object is unique in an execution, but it can change across executions. Thus, iGoodlock cannot use the object address to report a lock or a thread. Instead, it uses an abstraction computed out of static program information to identify a thread or a lock object. For example, the label of a statement at which an object is created could be used as its abstraction. We describe two better (i.e. more precise) object abstractions in Section 7.3. In this section, we assume that `abs`(o) returns some abstraction of the object o. Thus, for the deadlock cycle $\langle (t_1, L_1, l_1, C_1), (t_2, L_2, l_2, C_2), \ldots, (t_m, L_m, l_m, C_m) \rangle$, iGoodlock reports the cycle as $\langle$(`abs`(t_1), `abs`(l_1), C_1), (`abs`(t_2), `abs`(l_2), C_2), ..., (`abs`(t_m), `abs`(l_m), C_m)$\rangle$ to DeadlockFuzzer.

Given a potential deadlock by iGoodlock, DeadlockFuzzer executes the program using a random scheduler. A simple randomized execution algorithm is shown in Algorithm 8. Starting from the initial state s_0, this algorithm, at every state, randomly picks an enabled thread and executes its next statement. The algorithm terminates when the system reaches a state that has no enabled threads. At termination, if there is at least one thread that is alive, the algorithm reports a system stall.

A key limitation of this simple random scheduling algorithm is that it may not create real deadlocks very often. DeadlockFuzzer biases the random scheduler so that potential

Algorithm 8 simpleRandomChecker(s_0)

1: **INPUTS:** the initial state s_0
2: $s \Leftarrow s_0$
3: **while** `Enabled`$(s) \neq \emptyset$ **do**
4: $\quad$ $t \Leftarrow$ a random thread in `Enabled`(s)
5: $\quad$ $s \Leftarrow$ `Execute`(s, t)
6: **end while**
7: **if** `Alive`$(s) \neq \emptyset$ **then**
8: $\quad$ **print** 'System Stall!'
9: **end if**

deadlock cycles reported by iGoodlock get created with high probability. The active random deadlock checking algorithm is shown in Algorithm 9. Specifically, the algorithm takes an initial state s_0 and a potential deadlock cycle `Cycle` as inputs. It then executes the multithreaded program using the simple random scheduler, except that it performs some extra work when it encounters a lock acquire or lock release statement. If a thread t is about to acquire a lock l in the context C, then if (`abs`(t), `abs`(l), C) is present in `Cycle`, the scheduler pauses thread t before t acquires lock l, giving a chance to another thread, which is involved in the potential deadlock cycle, to acquire lock l subsequently. This ensures that the system creates the potential deadlock cycle `Cycle` with high probability.

Algorithm 9 maintains three data structures: `LockSet` that maps each thread to a stack of locks that are currently held by the thread, `Context` that maps each thread to a stack of statement labels where the thread has acquired the currently held locks, and `Paused` which is a set of threads that has been paused by DeadlockFuzzer. `Paused` is initialized to an empty set, and `LockSet` and `Context` are initialized to map each thread to an empty stack.

DeadlockFuzzer runs in a loop until there is no enabled thread. At termination, DeadlockFuzzer reports a system stall if there is at least one active thread in the execution. In each iteration of the loop, DeadlockFuzzer picks a random thread t that is enabled but not in the `Paused` set. If the next statement to be executed by t is not a lock acquire or release, t executes the statement and updates the state as in the simple random scheduling algorithm (see Algorithm 8). If the next statement to be executed by t is c: `Acquire`(l), c and l are pushed to `Context`$[t]$ and `LockSet`$[t]$, respectively. DeadlockFuzzer then checks if the acquire of l by t could lead to a deadlock using `checkRealDeadlock` in Algorithm 10. `checkRealDeadlock` goes over the current lockset of each thread and sees if it can find a cycle. If a cycle is discovered, then DeadlockFuzzer has created a *real* deadlock. If there is no cycle, then DeadlockFuzzer determines if t needs to be paused in order to get into a deadlock state. Specifically, it checks if (`abs`(t), `abs`(l), `Context`$[t]$) is present in `Cycle`. If t is added to `Paused`, then we pop from both `LockSet`$[t]$ and `Context`$[t]$ to reflect the fact that t has not really acquired the lock l. If the next statement to be executed by t is c: `Release`(l), then we pop from both `LockSet`$[t]$ and `Context`$[t]$.

At the end of each iteration, it may happen that the set `Paused` is equal to the set of

Algorithm 9 `DeadlockFuzzer`(s_0,`Cycle`)

1: **INPUTS:** the initial state s_0, a potential deadlock cycle `Cycle`
2: $s \Leftarrow s_0$
3: `Paused` $\Leftarrow \emptyset$
4: `LockSet` and `Context` map each thread to an empty stack
5: **while** `Enabled`(s) $\neq \emptyset$ **do**
6: $t \Leftarrow$ a random thread in `Enabled`(s)$\setminus$ `Paused`
7: `Stmt` $\Leftarrow$ next statement to be executed by t
8: **if** `Stmt` = c: `Acquire`(l) **then**
9: push l to `LockSet`[t]
10: push c to `Context`[t]
11: `checkRealDeadlock(LockSet)` // see **Algorithm 10**
12: **if** ((`abs`(t), `abs`(l), `Context`[t]) $\notin$ `Cycle`) **then**
13: $s \Leftarrow$ `Execute`(s,t)
14: **else**
15: pop from `LockSet`[t]
16: pop from `Context`[t]
17: add t to `Paused`
18: **end if**
19: **else if** `Stmt` = c: `Release`(l) **then**
20: pop from `LockSet`[t]
21: pop from `Context`[t]
22: $s \Leftarrow$ `Execute`(s,t)
23: **else**
24: $s \Leftarrow$ `Execute`(s,t)
25: **end if**
26: **if** |`Paused`| = |`Enabled`(s)| **then**
27: remove a random thread from `Paused`
28: **end if**
29: **end while**
30: **if** `Alive`(s) $\neq \emptyset$ **then**
31: **print** 'System Stall!'
32: **end if**

Algorithm 10 `checkRealDeadlock(LockSet)`

1: **INPUTS:** `LockSet` mapping each thread to its current stack of locks
2: **if** there exist distinct $t_1, t_2, \ldots, t_m$ and $l_1, l_2, \ldots, l_m$ such that l_m appears before l_1 in `LockSet`[t_m] and for each $i \in [1, m-1]$, l_i appears before l_{i+1} in `LockSet`[t_i] **then**
3: **print** 'Real Deadlock Found!'
4: **end if**

all enabled threads. This results in a state where DeadlockFuzzer has unfortunately paused all the enabled threads and the system cannot make any progress. We call this *thrashing.* DeadlockFuzzer handles this situation by removing a random thread from the set `Paused`. A thrash implies that DeadlockFuzzer has paused a thread in an unsuitable state. DeadlockFuzzer should avoid thrashing as much as possible in order to guarantee better performance and improve the probability of detecting real deadlocks.

7.3 Object abstractions

A key requirement of DeadlockFuzzer is that it should know where a thread needs to be paused, i.e. it needs to know if a thread t that is trying to acquire a lock l in a context C could lead to a deadlock. DeadlockFuzzer gets this information from iGoodlock, but this requires us to identify the lock and thread objects that are the "same" in the iGoodlock and DeadlockFuzzer executions. This kind of correlation cannot be done using the address (i.e. the unique id) of an object because object addresses change across executions. Therefore, we propose to use object abstraction—if two objects are same across executions, then they have the same abstraction. We assume $\texttt{abs}(o)$ computes the abstraction of an object.

There could be several ways to compute the abstraction of an object. One could use the label of the statement that allocated the object (i.e. the allocation site) as its abstraction. However, that would be too coarse-grained to distinctly identify many objects. For example, if one uses the factory pattern to allocate all thread objects, then all of the threads will have the same abstraction. Therefore, we need more contextual information about an allocation site to identify objects at finer granularity.

Note that if we use a coarse-grained abstraction, then DeadlockFuzzer will pause unnecessary threads before they try to acquire some unnecessary locks. This is because all these unnecessary threads and unnecessary locks might have the same abstraction as the relevant thread and lock, respectively. This will in turn reduce the effectiveness of our algorithm as DeadlockFuzzer will more often remove a thread from the `Paused` set due to the unavailability of any enabled thread. Note that we call this situation *thrashing*. Our experiments (see Chapter 9) show that if we use the trivial abstraction, where all objects have the same abstraction, then we get a lot of thrashing. This in turn reduces the probability of creating a real deadlock. On the other hand, if we consider too fine-grained abstraction for objects, then we will not be able to tolerate minor differences between two executions, causing threads to pause at fewer locations and miss deadlocks. We next describe two abstraction techniques for objects that we have found effective in our experiments.

7.3.1 Abstraction based on k-object-sensitivity

Given a multithreaded execution and a $k > 0$, let $o_1, \ldots, o_k$ be the sequence of objects such that for all $i \in [1, k-1]$, o_i is allocated by some method of object o_{i+1}. We define $\texttt{abs}^O_k(o_1)$

as the sequence $\langle c_1, \ldots, c_k \rangle$ where c_i is the label of the statement that allocated o_i. $\texttt{abs}^O_k(o_1)$ can then be used as an abstraction of o_1. We call this *abstraction based on k-object-sensitivity* because of the similarity to k-object-sensitive static analysis [83].

In order to compute $\texttt{abs}^O_k(o)$ for each object o during a multithreaded execution, we maintain a map `CreationMap` that maps each object o to a pair (o', c) if o is created by a method of object o' at the statement labeled c. This gives the following straightforward runtime algorithm for computing `CreationMap`.

- If a thread t executes the statement c: $o = \texttt{new}\ (o', T)$, then add $o \mapsto (o', c)$ to `CreationMap`.

One can use `CreationMap` to compute $\texttt{abs}^O_k(o)$ using the following recursive definition:

$$\texttt{abs}^O_k(o) = \langle\rangle \qquad \text{if } k = 0 \text{ or } \texttt{CreationMap}[o] = \bot$$

$$\texttt{abs}^O_{k+1}(o) = c :: \texttt{abs}^O_k(o') \qquad \text{if } \texttt{CreationMap}[o] = (o', c)$$

When an object is allocated inside a static method, it will not have a mapping in `CreationMap`. Consequently, $\texttt{abs}^O_k(o)$ may have fewer than k elements.

7.3.2 Abstraction based on light-weight execution indexing

Given a multithreaded execution, a $k > 0$, and an object o, let $m_n, m_{n-1}, \ldots, m_1$ be the call stack when o is created, i.e. o is created inside method m_1 and for all $i \in [1, n-1]$, m_i is called from method m_{i+1}. Let us also assume that c_{i+1} is the label of the statement at which m_{i+1} invokes m_i and q_{i+1} is the number of times m_i is invoked by m_{i+1} in the context $m_n, m_{n-1}, \ldots, m_{i+1}$. Then $\texttt{abs}^I_k(o)$ is defined as the sequence $[c_1, q_1, c_2, q_2, \ldots, c_k, q_k]$, where c_1 is the label of the statement at which o is created and q_1 is the number of times the statement is executed in the context $m_n, m_{n-1}, \ldots, m_1$.

```
1   main() {
2     for (int i=0; i<5; i++)
3       foo();
4   }
5   void foo() {
6     bar();
7     bar();
8   }
9   void bar() {
10    for (int i=0; i<3; i++)
11      Object l = new Object();
12  }
```

For example in the above code, if o is the first object created by the execution of `main`, then $\texttt{abs}^I_3(o)$ is the sequence $[11, 1, 6, 1, 3, 1]$. Similarly, if o is the last object created by the execution of `main`, then $\texttt{abs}^I_3(o)$ is the sequence $[11, 3, 7, 1, 3, 5]$. The idea of computing this kind of abstraction is similar to the idea of execution indexing proposed in [113], except that we ignore branch statements and loops. This makes our indexing light-weight, but less precise.

In order to compute $\texttt{abs}_k^I(o)$ for each object o during a multithreaded execution, we maintain a thread-local scalar d to track the depths of method calls and object creation statements and two thread-local maps `CallStack` and `Counters`. We use $\texttt{CallStack}_t$ to denote the `CallStack` map of thread t, and $\texttt{Counters}_t$ to denote the `Counters` map of thread t. The above data structures are updated at runtime as follows.

- Initialization:
  - for all t, $d_t \Leftarrow 0$
  - for all t and c, $\texttt{Counters}_t[d_t][c] \Leftarrow 0$
- If a thread t executes the statement c: $\texttt{Call}(o, m)$
  - $\texttt{Counters}_t[d_t][c] \Leftarrow \texttt{Counters}_t[d_t][c] + 1$
  - push c to $\texttt{CallStack}_t$
  - push $\texttt{Counters}_t[d_t][c]$ to $\texttt{CallStack}_t$
  - $d_t \Leftarrow d_t + 1$
  - for all c, $\texttt{Counters}_t[d_t][c] \Leftarrow 0$
- If a thread t executes the statement c: $\texttt{Return}(m)$
  - $d_t \Leftarrow d_t - 1$
  - pop twice from $\texttt{CallStack}_t$
- If a thread t executes the statement c: $o = \texttt{new}(o', T)$
  - $\texttt{Counters}_t[d_t][c] \Leftarrow \texttt{Counters}_t[d_t][c] + 1$
  - push c to $\texttt{CallStack}_t$
  - push $\texttt{Counters}_t[d_t][c]$ to $\texttt{CallStack}_t$
  - $\texttt{abs}_k^I(o)$ is the top $2k$ elements of $\texttt{CallStack}_t$
  - pop twice from $\texttt{CallStack}_t$

Note that $\texttt{abs}_k^I(o)$ has $2k$ elements, but if the call stack has fewer elements, then $\texttt{abs}_k^I(o)$ returns the full call stack.

7.3.3 Example

Consider again the example in Figure 4.2. To illustrate the utility of thread and lock abstractions, we uncomment the lines at 24 and 27. Now we create a third lock `o3` and a third thread which acquires `o2` and `o3` in order. iGoodlock as before will report the same deadlock cycle as in Section 7.1. In DeadlockFuzzer, if we do not use thread and lock abstractions, then with probability 0.5 (approx), the third thread will pause before acquiring the lock at line 16. This is because, without any knowledge about threads and locks involved in a potential

deadlock cycle, DeadlockFuzzer will pause any thread that reaches line 16. Therefore, if the third thread pauses before line 16, then the second thread will not be able to acquire lock `o2` at line 15 and it will be blocked. DeadlockFuzzer will eventually pause the first thread at line 16. At this point two threads are paused and one thread is blocked. This results in a *thrashing* (see Section 7.2). To get rid of this stall, DeadlockFuzzer will "un-pause" the first thread with probability 0.5 and we will miss the deadlock with probability 0.25 (approx). On the other hand, if we use object abstractions, then DeadlockFuzzer will never pause the third thread at line 16 and it will create the real deadlock with probability 1. This illustrates that if we do not use abstractions, then we get more thrashings and the probability of creating a real deadlock gets reduced.

7.4 Optimization: avoiding another potential cause for thrashing

We showed that using object and thread abstractions helps reduce thrashing; this in turn helps increase the probability of creating a deadlock. We show another key reason for a lot of thrashings using the following example and propose a solution to partly avoid such thrashings.

```
1: thread1{
2:   synchronized(l1){
3:     synchronized(l2){
4:     }
5:   }
6: }
```

```
 8: thread2{
 9:   synchronized(l1){
10:
11:   }
12:   synchronized(l2){
13:     synchronized(l1){
14:     }
15:   }
16: }
```

The above code avoids explicit thread creation for simplicity of exposition. iGoodlock will report a potential deadlock cycle in this code. In the active random deadlock checking phase, if `thread1` is paused first (at line 3) and if `thread2` has just started, then `thread2` will get blocked at line 9 because `thread1` is holding the lock `l1` and it has been paused and `thread2` cannot acquire the lock. Since we have one paused and one blocked thread, we get a thrashing. DeadlockFuzzer will "un-pause" `thread1` and we will miss the real deadlock. This is a common form of thrashing that we have observed in our benchmarks (Chapter 9).

In order to reduce the above pattern of thrashing, we make a thread to yield to other threads before it starts entering a deadlock cycle. Formally, if $(\texttt{abs}(t), \texttt{abs}(l), C)$ is a component of a potential deadlock cycle, then DeadlockFuzzer will make any thread t' with $\texttt{abs}(t) = \texttt{abs}(t')$ yield before a statement labeled c where c is the bottom-most element in the stack C. For example, in the above code, DeadlockFuzzer will make `thread1` yield before it tries to acquire lock `l1` at line 2. This will enable `thread2` to make progress (i.e.

acquire and release `l1` at lines 9 and 11, respectively). `Thread2` will then yield to any other thread before acquiring lock `l2` at line 12. Therefore, the real deadlock will get created with probability 1.

7.5 Confirming other concurrency errors

Algorithm 11 Checker(s_0, set of events *breakpoints*)

1: $s \Leftarrow s_0$
2: Paused $\Leftarrow \emptyset$
3: Initialize Σ, the local state used by the underlying analysis
4: **while** Enabled(s) $\neq \emptyset$ **do**
5: $t \Leftarrow$ a random thread in Enabled(s) $\setminus$ Paused
6: $e \Leftarrow$ next event to execute in t
7: Σ = analyze(Σ, e)
8: **if** $e \in$ *breakpoints* **then**
9: Paused $\Leftarrow$ check(e, Paused)
10: **end if**
11: **if** $e \notin$ Paused **then**
12: $s \Leftarrow$ Execute(s, e)
13: **end if**
14: **if** $|$Paused$| = |$Enabled(s)$|$ **then**
15: remove a random event from Paused
16: **end if**
17: **end while**
18: **if** Alive(s) $\neq \emptyset$ **then**
19: print ‘System stall!’
20: **end if**

Algorithm 11 generalizes the DeadlockFuzzer algorithm, and shows how we can confirm other classes of errors [65]. We assume that the potential error that we are trying to confirm is given in the form of a set of events (*breakpoints*) whose execution should be paused to exhibit the error. For example, in DeadlockFuzzer, the set of events is the set of tuples in the given deadlock cycle. The program under test is executed by randomly picking an enabled thread t that is not paused at each step of execution. The information regarding the next event e to execute in t is used to update the local state Σ used by the algorithm in `analyze`. For example, in DeadlockFuzzer, the event e is used to update the `LockSet` and `Context` used by DeadlockFuzzer (Algorithm 9). If e is in the set of events provided to the checker algorithm, then it might be paused in `check` to force the execution to reach a state that exhibits the potential error being considered. In DeadlockFuzzer, if the context of e matches that of a tuple in the given deadlock cycle, then it will be paused if permitted by the optimization in Section 7.4. Before pausing an event, the algorithm also checks to see if the given error has already been exhibited by the events that have been

paused thus far. If the error has been reproduced, the algorithm classifies the error as a real error and exits. If all enabled threads have been paused, one of them is randomly chosen and woken up in order to allow the program execution to make progress.

We have seen how real deadlocks and other concurrency errors can be confirmed in this chapter. In the next two chapters, we explain how we have implemented the predictive techniques into prototype tools, and also evaluate various aspects of our tools.

Chapter 8

Implementation

We explain how we have architected the prototype tools that implement our predictive techniques. All the tools can be broken down into two main pieces: the instrumentor and the analysis. The instrumentor adds hooks to the program under consideration so that it can observe events of interest (*e.g.*, lock acquires and releases, thread starts and joins) when the program executes, and pass those events and their context information on to the analysis. We describe the instrumentor and the analysis in our tools in the following sections. Our prototype tools are built for Java programs.

8.1 The instrumentor

We instrument the Java bytecode of the given program to insert probes so that we can collect context information regarding relevant events during program execution. Bytecode instrumentation allows us to analyze any Java program for which the source code is not available. We use two different frameworks to instrument: the Soot compiler framework [102] and JChord [1] which is a program analysis framework for Java. All our tools use Soot except for CheckMate (Chapter 5) which uses JChord. The probes inserted by instrumentation invoke methods (also implemented in Java) that implement the analysis in the tool. For example, in iGoodlock (Chapter 4), we are interested in tracking lock acquires and releases to compute potential deadlock cycles. Thus, we instrument bytecode to add hooks that track the execution of lock acquires and releases. The hook for lock acquires can call a method (say `lockAcq`()) and pass on the relevant lock, thread, and location information so that the method can update the lockset and context for the current thread and the lock dependency relation according to the algorithm described in Section 4.3.1. The hook for lock releases can similarly call another method (say `lockRel`()) that pops locks and location information from appropriate stacks according to the same algorithm. If we consider the example in Figure 4.2 shown in Chapter 4, then following is the sequence of method calls that would be called by the inserted probes during the non-deadlocking execution in which the second thread acquires and releases all the locks before the first thread can acquire any lock.

```
lockAcq(o2, T2, 15);
lockAcq(o1, T2, 16);
lockRel(o1, T2);
lockRel(o2, T2);
lockAcq(o1, T1, 15);
lockAcq(o2, T1, 16);
lockRel(o2, T1);
lockRel(o1, T1);
```

The methods called by probes would update the lock dependency relation as the program executes, and would compute the deadlock cycles in the end. The parameters o1, o2, T1, and T2 are unique integer identifiers for the locks and the threads in the program, and 15 and 16 are the source locations where the locks were acquired. The instrumentor assigns a unique identifier to each object during execution. The identifier for the same object however might be different in two different executions. Thus, we also use object abstractions (Chapter 7) built from static program information to identify objects across executions. To build the abstractions, in addition to tracking lock acquires and releases, we also have to insert probes to track object creation, and method calls and returns. Similarly, for other predictive techniques, we insert probes to track the execution of the statements relevant to the analyses in those techniques.

8.2 Implementing analyses

After the relevant statements have been instrumented by inserting probes, the analysis that we want to implement can be implemented in the methods that are called by the probes. The probes can pass on the dynamic (or static) information that the methods might need to implement the analysis. For example, the probes pass on the lock and thread identifiers and location information for iGoodlock (Chapter 4). In CheckMate (Chapter 5), the methods called by the probes generate the trace program (Algorithm 2) using the object identifiers and source location information as the program under test executes. In PRETEX (Chapter 6), the probes help to perform object race detection (Section 6.2.2) and generation and traversal of the computation lattice (Section 6.2.4). DeadlockFuzzer (Chapter 7) uses the information passed on by the probes to compute object abstractions and identify threads and locks potentially involved in a deadlock and to control the thread scheduler.

There are two concepts that are used a lot by our tools, and also often by other tools for concurrent programs: locksets and vector clocks (Chapter 3). We implement locksets by tracking lock acquires and releases, and vector clocks by tracking thread starts and joins and waits and notifies. The implementations of locksets and vector clocks have been factored out, and is shared across the tools that employ them. There are some tool specific optimizations and details that we provide in the rest of the section.

In CheckMate, given a Java program, we first use the `ConditionAnnotation` class (Figure 5.8) to manually annotate the predicate associated with each condition variable in the program. CheckMate then uses JChord [1] to instrument lock acquires and releases,

waits and notifies, thread starts and joins, and all writes to objects and method calls in the program. It then executes the annotated, instrumented program on given input data and generates the trace program. Finally, it uses the JPF model checker [54] to explore all possible executions of the trace program and report deadlocks.

In PRETEX, while performing object race detection in the first stage of the analysis, we track method invocations on objects. The number of method invocations in an execution can often be very large, and saving information for each of them can stress the memory requirement of the program under test. Thus, we implement the following optimization. Before adding a method invocation event to the database of events, we first search the database to see if such an event already exists. If it does, then we do not add the current event to the database, or else we add it to the database. Since we track the thread dependencies arising out of `start()`, `join()`, `wait()`, `notify()`, and `notifyAll()` and ignore other dependencies present between threads, a considerable number of events that occurs on an object in a thread occurs with the same vector clock and lockset. Therefore, for all of these events, we have to add only a single entry to the database.

For the second stage of PRETEX, we use an off-the-shelf PFSA builder [92] to generate likely typestate automata for object types that we found to be in race in the first stage. The PFSA builder takes a list of method call sequences as input, and outputs an automaton that accepts all of those sequences and rejects most of the others. In the third stage, for each automaton that we generate in the previous stage, we instrument the program at each point where a method call present in one of the edges of the automaton is invoked. The instrumentation uses the method invocation events to build the levels of the multithreaded computation lattice.

DeadlockFuzzer can go into livelocks while pausing threads to reproduce a potential deadlock in a program. Livelocks happen when all threads of the program end up being paused, except for one thread that does something in a loop without synchronizing with other threads. In order to avoid livelocks, we create a monitor thread that periodically "un-pauses" those threads that are paused for a long time.

The source code for iGoodlock and DeadlockFuzzer can be found at `http://srl.cs.berkeley.edu/~ksen/calfuzzer/`. The source code for CheckMate is included with the source code for JChord that can be downloaded from `http://pag.gatech.edu/chord/`. The source code for PRETEX can be found at `https://github.com/pallavij/PRETEX`. In the next chapter, we evaluate various aspects of our tools on a number of real-world programs.

Chapter 9

Evaluation

We have implemented the predictive analyses presented in the previous chapters into prototype tools for Java programs, and have experimented the tools with a number of real-world Java benchmarks to evaluate their efficiency and effectiveness. We discuss our experiments and findings in the subsequent sections.

9.1 Resource deadlocks

We evaluated iGoodlock (Chapter 4) and CheckMate (Chapter 5) on a variety of Java programs and libraries. We ran our experiments on a dual socket Intel Xeon 2GHz quad core server with 8GB of RAM. The following programs were included in our benchmarks: cache4j, a fast thread-safe implementation of a cache for Java objects; sor, a successive over-relaxation benchmark, and hedc, a web-crawler application, both from ETH [108]; jspider, a highly configurable and customizable Web Spider engine; and Jigsaw, W3C's leading-edge Web server platform. We ran CheckMate on more benchmarks in which we found communication deadlocks, but discuss those benchmarks in Section 9.2. We created a test harness for Jigsaw that concurrently generates simultaneous requests to the web server, simulating multiple clients, and administrative commands (such as "shutdown server") to exercise the multithreaded server in a highly concurrent situation.

The libraries we experimented on include synchronized lists and maps from the Java Collections Framework, Java logging facilities (`java.util.logging`), and the Swing GUI framework (`javax.swing`). Another widely used library included in our benchmarks is the Database Connection Pool (DBCP) component of the Apache Commons project. We created general test harnesses to use these libraries with multiple threads. For example, to test the Java Collections in a concurrent setting, we used the synchronized wrappers in `java.util.Collections`.

Program name	Lines of code	Avg. Runtime in msec.			# Deadlock cycles	
		Normal	iGoodlock	CM	iGoodlock	Real
cache4j	3,897	2,045	3,409	5,890	0	0
sor	17,718	163	396	3,000	0	0
hedc	25,024	165	1,668	47,000	0	0
jspider	10,252	4,622	5,020	6,000	0	0
Jigsaw	160,388	-	-	-	283	≥ 29
Java Logging	4,248	166	272	5,700	3	3
Java Swing	337,291	4,694	9,563	1,22,600	1	1
DBCP	27,194	603	1,393	15,300	2	2
Synchronized Lists (ArrayList, Stack, LinkedList)	17,633	2,862	3,244	4,000	9 + 9 + 9	9 + 9 + 9
Synchronized Maps (HashMap, TreeMap, WeakHashMap, LinkedHashMap, IdentityHashMap)	18,911	2,295	2,596	3,000	4 + 4 + 4 + 4 + 4	4 + 4 + 4 + 4 + 4

Table 9.1: Experimental results for resource deadlocks

9.1.1 Results

Table 9.1 shows the results for resource deadlocks. The second column reports the number of lines of source code that was instrumented for iGoodlock. If the program uses libraries that are also instrumented, they are included in the count. We instrument more source lines for CheckMate since we also track wait-notify synchronization in CheckMate. We give details about CheckMate's instrumentation in Section 9.2. The third column shows the average runtime of a normal execution of the program without any instrumentation or analysis. The fourth column is the runtime of iGoodlock. The fifth column is the average runtime of CheckMate. For CheckMate, we include the total time it takes to run the instrumented program to generate the trace program and to model check the trace program. We give details regarding the trace programs generated and the model checking process in Section 9.2. The table shows that the overhead of iGoodlock is within a factor of three for all programs except hedc. Note that runtime for the web server Jigsaw is not reported due to its interactive nature. The runtime for CheckMate is much more – it can be as large as 30X of the time taken by iGoodlock. Thus, iGoodlock has better runtime complexity as compared to CheckMate, but it can only find resource deadlocks. The sixth column is the number of potential deadlocks reported by iGoodlock. The last column reports the real deadlocks that we found after manual inspection. For Jigsaw, we could confirm 29 of the cycles using DeadlockFuzzer(Section 9.4), and thus, we can say for sure that Jigsaw has 29 or more real deadlocks. Note that two distinct deadlock cycles might correspond to the same root bug, that is, the same fix might resolve both of them. This can happen when, say, different locks are involved in the deadlock cycles, but those locks are acquired at the

```
org.w3c.jigsaw.http.httpd {
 384:   SocketClientFactory factory;
1442:   void cleanup(...) {
1455:           factory.shutdown();}
1711:   void run() {
1734:           cleanup(...);}}

org.w3c.jigsaw.http.socket.SocketClient {
 42:    SocketClientFactory pool;
111:    void run() {
152:            pool.clientConnectionFinished(...);}}

org.w3c.jigsaw.http.socket.SocketClientFactory {
130:    SocketClientState csList;
574:    synchronized boolean decrIdleCount() {...}
618:    boolean clientConnectionFinished(...) {
623:            synchronized (csList) {
626:                    decrIdleCount();}}
867:    synchronized void killClients(...) {
872:            synchronized (csList) {...}}
902:    void shutdown() {
904:                    killClients(...);}
}
```

Figure 9.1: Deadlock in Jigsaw

same statements. CheckMate found all the resource deadlocks found by iGoodlock, and its evaluation is presented in more detail in Section 9.2.

9.1.2 Deadlocks found

Both iGoodlock and CheckMate found a number of previously unknown and known resource deadlocks in our benchmarks. We next describe some of them.

Two previously unknown deadlocks were found in Jigsaw. As shown in Figure 9.3, when the http server shuts down, it calls cleanup code that shuts down the `SocketClientFactory`. The shutdown code holds a lock on the factory at line 867, and in turn attempts to acquire the lock on `csList` at line 872. On the other hand, when a `SocketClient` is closing, it also calls into the factory to update a global count. In this situation, the locks are held in the opposite order: the lock on `csList` is acquired first at line 623, and then on the factory at line 574. Another similar deadlock occurs when a `SocketClient` kills an idle connection. These also involve the same locks, but are acquired at different program locations.

The deadlock in the Java Swing benchmark occurs when a program synchronizes on a `JFrame` object, and invokes the `setCaretPosition()` method on a `JTextArea` object that is a member of the `JFrame` object. The sequence of lock acquires that leads to the deadlock is

as follows. The `main` thread obtains a lock on the `JFrame` object, and an `EventQueue` thread which is also running, obtains a lock on a `BasicTextUI$BasicCaret` object at line number 1304 in *javax/swing/text/DefaultCaret.java*. The `main` thread then tries to obtain a lock on the `BasicTextUI$BasicCaret` object at line number 1244 in *javax/swing/text/DefaultCaret.java*, but fails to do so since the lock has not been released by the `EventQueue` thread. The `EventQueue` thread tries to acquire the lock on the `JFrame` object at line number 407 in *javax/swing/RepaintManager.java* but cannot since it is still held by the `main` thread. The program goes into a deadlock. This deadlock corresponds to a bug that has been reported at `http://bugs.sun.com/view_bug.do?bug_id=4839713`.

One of the deadlocks that we found in the DBCP benchmark occurs when a thread tries to create a `PreparedStatement`, and another thread simultaneously closes another `PreparedStatement`. The sequence of lock acquires that exhibits this deadlock is as follows. The first thread obtains a lock on a `Connection` object at line number 185 in *org/apache/commons/dbcp/DelegatingConnection.java*. The second thread obtains a lock on a `KeyedObjectPool` object at line number 78 in *org/apache/commons/dbcp/PoolablePreparedStatement.java*. The first thread then tries to obtain a lock on the same `KeyedObjectPool` object at line number 87 in *org/apache/commons/dbcp/PoolingConnection.java*, but cannot obtain it since it is held by the second thread. The second thread tries to obtain a lock on the `Connection` object at line number 106 in *org/apache/commons/dbcp/PoolablePreparedStatement.java*, but cannot acquire it since the lock has not yet been released by the first thread. The program, thus, goes into a deadlock.

The deadlocks in the Java Collections Framework happen when multiple threads are operating on shared collection objects wrapped with the `synchronizedX` classes. For example, in the `synchronizedList` classes, the deadlock can happen if one thread executes `l1.addAll(l2)` concurrently with another thread executing `l2.retainAll(l1)`. There are three methods, `addAll()`, `removeAll()`, and `retainAll()` that obtain locks on both `l1` and `l2` for a total of 9 combinations of deadlock cycles. The `synchronizedMap` classes have 4 combinations with the methods `equals()` and `get()`.

The test cases for Java Collections are artificial in the sense that the deadlocks in those benchmarks arise due to inappropriate use of the API methods. We used these benchmarks because they have been used by researchers in previous work (e.g. Williams et al. [112] and Jula et al. [69]), and we wanted to validate our tool against these benchmarks.

9.2 Communication deadlocks

We experimented CheckMate (Chapter 5) with several Java libraries and applications. We ran all our experiments on a dual socket Intel Xeon 2GHz quad core server with 8GB RAM. The libraries that we experimented with include the Apache log4j logging library (`log4j`), the Apache Commons Pool object pooling library (`pool`), an implementation of the OSGi framework (`felix`), the Apache Lucene text search library (`lucene`), and a reliable multicast communication library (`jgroups`). We used two different versions of `jgroups`. We

Program name	No. of annts	Orig prog LOC	Trace prog LOC	Orig prog time	Time to gen prog	JPF (orig prog)	JPF (trace prog)	No. err trcs	Pot. errs	Real errs	Kwn. errs
groovy-1.1	1	45,796	59	0.118s	1s	> 1h	1.3s	5	1/0	1/0	1/0
log4j-1.2.13	2	48,023	225	0.116s	1s	-	8.7s	167	2/0	1/0	1/0
pool-1.5 (harness 1)	4	48,024	136	0.116s	1s	> 1h	2.3s	41	1/0	1/0	1/0
pool-1.5 (harness 2)	4	48,024	191	0.123s	1s	> 1h	2.6s	36	1/0	1/0	1/0
felix-1.0.0	4	73,512	113	0.173s	2.8s	-	-	-	-	1/0	1/0
lucene-2.3.0 (harness 1)	9	68,311	298	0.230s	3s	> 1h	1s	0	0/0	0/0	1/0
lucene-2.3.0 (harness 2)	9	81,071	3,534	0.296s	3.6s	> 1h	20s	0	0/0	0/0	1/0
jgroups-2.6.1	12	92,934	118	0.228s	4s	-	3.4s	39	2/0	1/0	1/0
jigsaw-2.2.6	17	122,806	3,509	-	-	-	> 1h	7894	2/7	1/5	0/2
jruby-1.0.0	16	136,479	966	1.1s	13.7s	-	3.9s	58	1/0	1/0	1/0
jgroups-2.5.1	15	160,644	2,545	9.89s	21s	-	> 1h	124	1/0	0/0	1/0
java logging (jdk-1.5.0)	0	43,795	131	0.177s	2s	> 1h	3.7s	96	0/2	0/1	0/1
dbcp-1.2.1	0	90,821	400	0.74s	3.3s	-	12s	320	0/2	0/2	0/2
java swing (jdk-1.5.0)	0	264,528	1,155	0.96s	17.6s	-	105s	685	2/1	0/1	0/1

Table 9.2: Experimental results for CheckMate

also experimented with the following libraries that we also used for iGoodlock (Section 9.1): the Java logging library (`java.util.logging`), the Apache Commons DBCP database connection pooling library (`dbcp`), and the Java swing library (`javax.swing`). We wrote test harnesses exercising each library's API, including two different harnesses for each of `pool` and `lucene`, and a single harness for each of the remaining libraries.

The applications include Groovy, a Java implementation of a dynamic language that targets Java bytecode (`groovy`), and JRuby, a Java implementation of the Ruby programming language (`jruby`). We also used the `jigsaw` web server (Section 9.1).

9.2.1 Results

Table 9.2 summarizes our experimental results. The second column reports the number of `ConditionAnnotation`'s we had to provide, each annotating a different synchronization predicate in the benchmark. We report the number of `ConditionAnnotation`'s that we had to define, and not the total number of lines of code that we had to use to define the `ConditionAnnotation`'s and to invoke methods on those `ConditionAnnotation`'s. The numbers in this column show that the annotation burden of our approach is very small.

The third column shows the number of lines of Java code in methods that were executed in

the original program. The fourth column shows the number of lines of Java code in the trace program. Notice that the trace programs are much smaller than (executed parts of) original programs. Although the trace program unrolls all loops and inlines all methods executed in the original program, we use optimizations as explained in Section 5.3.2 to contain the size of the trace program.

The fifth column gives the average runtime of the original program without any instrumentation. We do not report the runtime for the `jigsaw` webserver because of its interactive nature. The sixth column gives the average runtime of the original program with annotations and instrumentation; it includes the time to generate the trace program. Comparing these two columns shows that the runtime overhead of CheckMate is acceptable.

The seventh column gives the average runtime of JPF on the original program. We could not run JPF on eight of these programs because it does not support some JDK libraries, and has limited support for reflection. For the remaining six programs, JPF did not terminate within 1 hour nor did it report any error traces.

The eighth column shows the average runtime of JPF on the trace programs. It terminates within a few seconds on eleven of these programs. It does not terminate within 1 hour for `jgroups-2.5.1` and `jigsaw-2.2.6`, but it reports a number of error traces in that time. These benchmarks have a lot of threads (31 for `jgroups-2.5.1` and 12 for `jigsaw-2.2.6`), hence a huge number of thread interleavings, which makes model checking slow. JPF crashes on the trace program for `felix-1.0.0`. Comparing the runtime of JPF on the original and trace programs shows that it is much more feasible to model check the trace programs.

The ninth column shows the number of error traces produced by JPF for the trace programs. An error trace is an interleaving of threads that leads to a deadlock. Not each error trace leads to a different deadlock, and thus, the number of error traces is not an indication of the number of different deadlocks in the program. Hence, CheckMate groups together error traces in which the same set of statements (either lock acquires or calls to `wait()`) is blocked, and reports each such group as a potential deadlock. The tenth column shows the number of these potential deadlocks reported by CheckMate. The eleventh column shows how many of these deadlocks we could manually confirm as real, and the final column shows the number of deadlocks that were previously known to us. The first number in each entry in the last three columns is the number of communication deadlocks including the deadlocks that involve both locks and condition variables. The second number is the number of resource deadlocks. In most of the benchmarks, we were able to find all previously known deadlocks. Since `JPF` crashed on the trace program for `felix-1.0.0`, we applied a randomized model checker (i.e. a model checker that tries out random thread schedules) to it. The randomized model checker reported a deadlock that was the same as its previously known communication deadlock.

9.2.2 Deadlocks found

We found a number of previously known and unknown deadlocks in our experiments. We discuss some of them in detail below. The example in Figure 5.4 documents a previously

known communication deadlock that we found in `log4j`. The deadlock is reported at `https://issues.apache.org/bugzilla/show_bug.cgi?id=38137`.

```
           T2
326:synch (writeLock) {
327:  concurrentReads++;
328:}
           T1
126:synch (writeLock) {
304:  synch (writeQueue) {
305:    while (concurrentReads != 0) {
307:       writeQueue.wait();
309:    }
310:  }
129:}
           T2
332:synch (writeLock) {
333:  concurrentReads--;
334:}
335:synch (writeQueue) {
336:  writeQueue.notify();
337:}
```

Figure 9.2: Deadlock in `groovy`.

Figure 9.2 shows a previously known deadlock in `groovy` reported at `http://jira.codehaus.org/browse/GROOVY-1890`. It shows relevant code from `MemoryAwareConcurrentReadMap.java`. This deadlock involves both locks and condition variables. Thread `T2` increments field `concurrentReads` of a `MemoryAwareConcurrentReadMap` object. Thread `T1` checks predicate `concurrentReads != 0`. Since this predicate is true, it executes the `wait()` on line 307. `T1` executes the `wait()` on `writeQueue`, but it also holds a lock on `writeLock`. Thread `T2` is the only thread that can wake it up, but before `T2` can reach the `notify()` on line 336, it needs to acquire the lock on `writeLock` to decrement the value of `concurrentReads`. Since the lock on `writeLock` is held by `T1`, it gets blocked. Thus, `T1` is waiting to be notified by `T2`, and `T2` is waiting for `T1` to release `writeLock`.

We found a previously unknown communication deadlock in `jigsaw`. Figure 9.3 explains the deadlock. The line numbers in the figure are of statements in `ResourceStoreManager.java` in the benchmark. Thread `T1` is a `StoreManagerSweeper` thread that executes the `wait()` on line 406 after it has been started. But, before it can execute this `wait()`, the server receives a request to shut down. Thread `T2`, which is a `httpd` server thread, tries to shut down the `StoreManagerSweeper` thread, and invokes `notifyAll()` at line 419 during the process of shutting down. This `notifyAll()` is the notification that is meant to wake `T1` up when it waits at the `wait()` on line 406. Thus,

```
        T1                                    T2
                                  417:synch void shutdown() {
                                  419:  notifyAll();
                                  420:}
401:boolean done = false;
404:while(!done) {
406:  wait();
407:  done = true;
410:}
```

Figure 9.3: Deadlock in `jigsaw`.

when `T1` actually executes the `wait()`, it just gets hung there. It has already missed the notification that was supposed to wake it up.

9.3 Typestate errors

We evaluated PRETEX on a number of benchmark programs. We ran our experiments on a laptop with a 2GHz Intel Core 2 Duo processor and 2GB RAM. We considered the following benchmark programs some of which we have already seen for deadlocks before : `hedc`, a meta-crawler application kernel developed at ETH [108]; `weblech`, a website download tool, `tornado`, a multithreaded web server; `cache4j`, a fast thread-safe implementation of a cache for Java objects; `jspider`, a web spider engine; `jigsaw`, W3C's web server, and apache ftpserver. The eighth column of Table 9.3 gives the lines of source code for these benchmarks. The last column in the table is the number of threads that were spawned for the benchmarks. All of these were closed programs except for `jigsaw` and `tornado`. As before for `jigsaw`, we wrote a harness that spawned a number of threads and queried the web server for different urls. We wrote a similar harness for the other web server `tornado`.

9.3.1 Results

Table 9.3 summarizes the average execution time of the various benchmarks for the different stages of PRETEX. The second column gives the average execution time of the unmodified benchmark. The third column is the average time taken for obtaining the method call sequences for objects of a particular type. The fourth column gives the average time for the PFSA builder to build a PFSA from a set of method call sequences. The fifth column is the average execution time to run the predictive typestate checker using a single automaton. All of the execution time is in milliseconds. The sixth column gives the total number of typestate errors reported by our tool. An error that is reported more than once is counted only once, and not the number of times it was reported. We manually inspect all the errors that are reported, and provide the number of real errors that we find in the seventh column.

Program	Normal Time	Time (Seqs)	Time (PFSA)	Time (Typestate)	Reported errs	Real errs	LOC	Thrds
tornado	4141	4125	1235	4140	6	0	1326	40
cache4j	4250	90421	1228	99609	3	0	3897	10
hedc	2813	2829	1352	2766	5	0	29948	5
weblech	1079	2641	1353	1609	6	1	35175	3
jspider	641	922	1233	781	7	0	64933	5
ftpserver	4890	8109	152	8125	8	0	127297	40
jigsaw	39031	39000	1352	39000	12	0	381348	30

Table 9.3: Execution time for typestate checking

As can be seen from the table, the execution time of an application after being instrumented to print the method call sequences is less than 3 times the execution time of the original application, except in the case of `cache4j`. The execution time of an application after being instrumented to predict typestate errors is less than 2 times the execution time of the uninstrumented application, except for `cache4j`. The overhead of instrumentation is thus very small. The PFSA builder takes almost constant time to build a PFSA from a set of method sequences. We manually examined the errors that were reported, and found one real error in weblech which we describe in the next section.

There are two main sources of false positives (a reported error not being a real error) in our experiments. Firstly, as in the other predictive analyses, the happens-before relation here is an approximation of the exact happens-before relation between events. As a result, we might consider an infeasible ordering of events. Secondly, since the automata that we build are based on the method call sequences that are observed during a certain execution of the program, they do not capture all legitimate method call sequences on objects of the type considered. Thus, some of the reported errors correspond to legitimate method call sequences that were not observed when the typestate automata were generated.

9.3.2 Typestate error found

In `weblech`, the URLs to be examined are queued in an instance of the class `DownloadQueue`. In `Spider.java`, due to insufficient synchronization, a thread might try to retrieve a URL from `DownloadQueue` even if there is no URL in it. The relevant portion of the code is shown below.

When `queue.getNextInQueue()` is called in the above code, the condition that `queueSize()` $\neq$ `0` could have become false. The execution of `queue.getNextInQueue()` can result in an exception being thrown if the size of `queue` is 0. The typestate automaton inferred for `DownloadQueue` by PRETEX correctly infers that `size()` should always be called before `getNextInQueue()`. The automaton is shown in figure 9.5. Using this typestate automaton we could predict the typestate error in the above code.

```
if (queueSize() == 0 && downloadsInProgress > 0)
{
    ....
    continue;
}
else if (queueSize() == 0)
{
    break;
}
....
synchronized (queue )
{
    nextURL = queue.getNextInQueue();
    downloadsInProgress++;
}
```

Figure 9.4: Typestate error in `weblech`

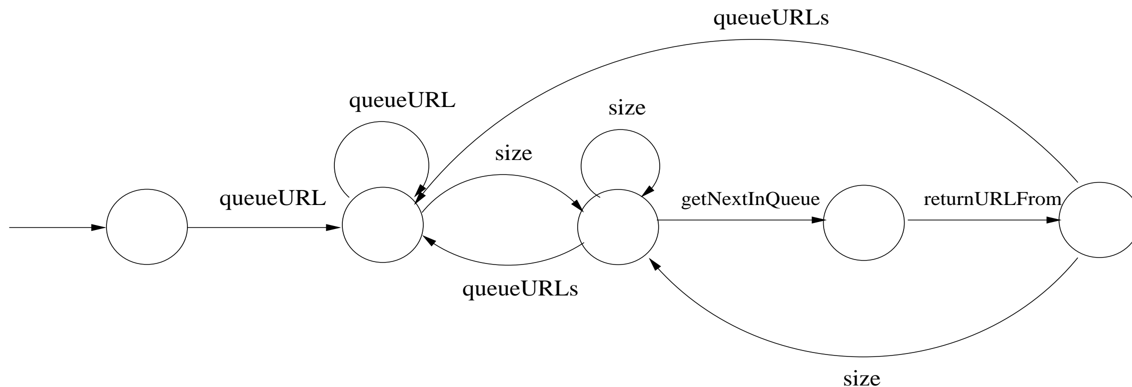


Figure 9.5: Typestate automaton inferred for DownloadQueue

9.4 Confirming Resource Deadlocks

We evaluated DeadlockFuzzer on the potential resource deadlocks reported by iGoodlock (Section 9.1). For each potential deadlock, we used DeadlockFuzzer to try to automatically reproduce it and confirm it. As for iGoodlock, we instrumented Java bytecode with Soot to observe relevant events and to control the thread scheduler.

9.4.1 Results

Table 9.4 shows the results of using DeadlockFuzzer. We used the same benchmarks as for iGoodlock (Section 9.1), and used iGoodlock to predict resource deadlocks in them. The third column shows the average runtime (in milliseconds) of a normal execution of the

Program name	Lines of code	Avg. Runtime		# Deadlock cycles			Prob.	No. of thrashes
		Normal	DF	iGL	Real	DF		
cache4j	3,897	2,045	-	0	0	-	-	-
sor	17,718	163	-	0	0	-	-	-
hedc	25,024	165	-	0	0	-	-	-
jspider	10,252	4,622	-	0	0	-	-	-
Jigsaw	160,388	-	-	283	$\geq$ 29	29	0.214	18.97
Java Logging	4,248	166	493	3	3	3	1.00	0.00
Java Swing	337,291	4,694	28,052	1	1	1	1.00	4.83
DBCP	27,194	603	1,393	2	2	2	1.00	0.00
Synchronized Lists (ArrayList, Stack, LinkedList)	17,633	2,862	7,070	9 + 9 + 9	9 + 9 + 9	9 + 9 + 9	0.99	0.0
Synchronized Maps (HashMap, TreeMap, WeakHashMap, LinkedHashMap, LinkedHashMap, IdentityHashMap)	18,911	2,295	2898	4 + 4 + 4 + 4 + 4	4 + 4 + 4 + 4 + 4	4 + 4 + 4 + 4 + 4	0.52	0.04

Table 9.4: Experimental results for DeadlockFuzzer. (Context + 2nd Abstraction + Yield optimization)

program without any instrumentation or analysis, and the fourth column is the average runtime of DeadlockFuzzer. The table shows that the overhead of DeadlockFuzzer is within a factor of six, even for large programs. Note that runtime for the web server Jigsaw is again not reported due to its interactive nature.

The fifth column is the number of potential deadlocks reported by iGoodlock. The sixth column is the number of cycles that correspond to real deadlocks after manual inspection. For Jigsaw, since DeadlockFuzzer could reproduce 29 deadlocks, we can say for sure that Jigsaw has 29 or more real deadlocks. With the exception of Jigsaw, iGoodlock was precise enough to report only real deadlocks. The seventh column is the number of deadlock cycles confirmed by DeadlockFuzzer. The eighth column is the empirical probability of DeadlockFuzzer reproducing the deadlock cycle. We ran DeadlockFuzzer 100 times for each cycle and calculated the fraction of executions that deadlocked using DeadlockFuzzer. Our experiments show that DeadlockFuzzer reproduces the potential deadlock cycles reported by iGoodlock with very high probability. We observed that for some Collections benchmarks, DeadlockFuzzer reported a low probability of 0.5 for creating a deadlock. After looking into the report, we found that in the executions where DeadlockFuzzer reported no deadlock, DeadlockFuzzer created a deadlock which was different from the potential deadlock cycle provided as input to DeadlockFuzzer. For comparison, we also ran each of the programs normally without instrumentation for 100 times to observe if these deadlocks could occur under normal testing. None of the runs resulted in

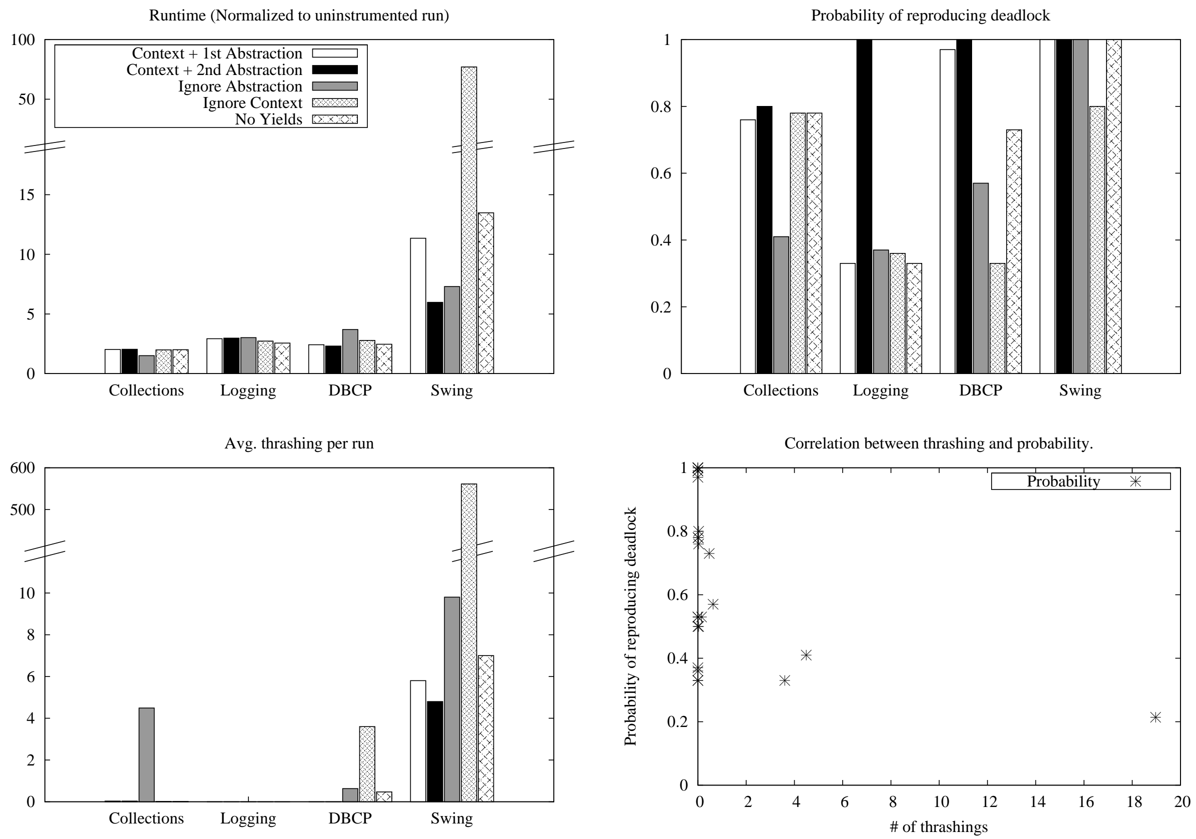


Figure 9.6: Performance and effectiveness of variations of DeadlockFuzzer

a deadlock, as opposed to a run with DeadlockFuzzer which almost always went into deadlock. Column 9 shows the average number of thrashings (pausing the wrong thread or at the wrong location) per run. Columns 8 and 9 show that the probability of creating a deadlock decreases as the number of thrashings increases.

We conducted additional experiments to evaluate the effectiveness of various design decisions for DeadlockFuzzer. We tried variants of DeadlockFuzzer: 1) with abstraction based on k-object-sensitivity, 2) with abstraction based on light-weight execution indexing, 3) with the trivial abstraction, 4) without context information, and 5) with the optimization in Section 7.4 turned off. Figure 9.6 summarizes the results of our experiments. Note that the results in Table 9.4 correspond to the variant 2, where we use the light-weight execution indexing abstraction, context information, and the optimization in Section 7.4. We found this variant to be the best performer: it created deadlocks with higher probability than any other variant and it ran efficiently with minimal number of thrashings.

The first graph shows the correlation between the various variants of DeadlockFuzzer and average runtime. The second graph shows the probability of creating a deadlock by the

variants of DeadlockFuzzer. The third graph shows the average number of thrashings encountered by each variant of DeadlockFuzzer. The fourth graph shows the correlation between the number of thrashings and the probability of creating a deadlock.

The first graph shows that variant 2, which uses execution indexing, performs better than variant 1, which uses k-object-sensitivity. The second graph shows that the probability of creating a deadlock is maximum for variant 2 on our benchmarks. The difference is significant for the Logging and DBCP benchmarks. Ignoring abstraction entirely (i.e. variant 3) led to a lot of thrashing in Collections and decreased the probability of creating a deadlock. The third graph on the Swing benchmark shows that variant 2 has minimum thrashing. Ignoring context information increased the thrashing and the runtime overhead for the Swing benchmark. In the Swing benchmark, the same locks are acquired and released many times at many different program locations during the execution. Hence, ignoring the context of lock acquires and releases leads to a huge amount of thrashing.

The first graph which plots average runtime for each variant shows some anomaly. It shows that variant 3 runs faster than variant 2 for Collections—this should not be true given that variant 3 thrashes more than variant 2. We found the following reason for this anomaly. Without the right debugging information provided by iGoodlock, it is possible for DeadlockFuzzer to pause at wrong locations but, by chance, introduce a real deadlock which is unrelated to the deadlock cycle it was trying to reproduce. This causes the anomaly in the first graph where the runtime overhead for Collections is lower when abstraction is ignored, but the number of thrashings is more. The runtime is measured as the time it takes from the start of the execution to either normal termination or when a deadlock is found. DeadlockFuzzer with our light-weight execution indexing abstraction faithfully reproduces the given cycle, which may happen late in the execution. For more imprecise variants such as the one ignoring abstraction, a deadlock early in the execution may be reproduced wrongfully, thus reducing the runtime.

The fourth graph shows that the probability of creating a deadlock goes down as the number of thrashings increases. This validates our claim that thrashings are not good for creating deadlocks with high probability and our variant 2 tries to reduce such thrashings significantly by considering context information and object abstraction based on execution indexing, and by applying the optimization in Section 7.4.

9.4.2 Incompleteness of DeadlockFuzzer

Since DeadlockFuzzer is not complete, if it does not classify a deadlock reported by iGoodlock as a real deadlock, we cannot definitely say that the deadlock is a false warning. For example, in the Jigsaw benchmark, the iGoodlock algorithm reported 283 deadlocks. Of these 29 were reported as real deadlocks by DeadlockFuzzer. We manually looked into the rest of the deadlocks to see if they were false warnings by iGoodlock, or real deadlocks that were not caught by DeadlockFuzzer. For 18 of the cycles reported, we can say with a high confidence that they are false warnings reported by the iGoodlock algorithm. These cycles involve locks that are acquired at the same program statements, but by different threads.

There is a single reason why all of these deadlocks are false positives. The deadlocks can occur only if a `CachedThread` invokes its `waitForRunner()` method before that `CachedThread` has been started by another thread. This is clearly not possible in an actual execution of Jigsaw. Since iGoodlock does not take the happens-before relation between lock acquires and releases into account, it reports these spurious deadlocks. For the rest of the cycles reported by iGoodlock, we cannot say with reasonable confidence if they are false warnings, or if they are real deadlocks that were missed by DeadlockFuzzer.

In the next chapter, we conclude our first section on testing concurrent programs by comparing our predictive techniques with other related work that has been done previously.

Chapter 10

Other Related Work

Plenty of work has been done on predictive analyses for finding bugs in multithreaded programs. But, most of this work focuses only on prediction of errors. We, on the other hand, not only predict errors, but also work towards automatically confirming the predicted errors. In addition, we have not only designed analyses for common classes of bugs like resource deadlocks that have been well-studied by previous work, but have also built techniques for harder-to-find but equally insidious bugs like communication deadlocks, hybrid between resource and communication deadlocks, and typestate errors. We have already explained some related work in previous chapters. In this chapter, we describe and compare against other related work that also come close to our direction of work.

10.1 Predictive testing

There has been previous work on predictive testing techniques on finding common classes of errors like data races [93, 32, 108, 24, 23], atomicity violations [109, 39, 110], and resource deadlocks [52, 53, 13, 6]. Our predictive testing techniques have the same flavor as these other predictive techniques in that they all predict bugs that could have occurred in a different thread interleaving that did not show up in any of the observed program executions. As in our work, these analyses also identify and track the relevant events and the synchronization observed during execution to predict potential bugs.

Apart from considering resource deadlocks which have been well-studied by previous work, we have designed predictive analyses for communication deadlocks, deadlocks that involve both locks and condition variables, and typestate errors for concurrent systems that have not received much attention in previous work. There is little prior work on detecting deadlocks involving condition variables. Agarwal and Stoller [5] dynamically predict missed notification deadlocks, in particular they define a happens-before ordering between synchronization events, and use it to reason if a wait that was woken up by a notify could have happened after that notify. Farchi et al [36] describe several concurrency bug patterns that occur in practice including missed notification. They also describe a heuristic that can

increase the probability of manifesting a missed notification during testing. Li et al [76] build a deadlock monitor that runs as a system daemon, and detects deadlocks that actually occur during the execution of systems with multiple processes or threads. The monitor can detect deadlocks involving semaphores and pipes in addition to locks. The monitor however cannot predict deadlocks in other executions. There has also been some work that use static analysis to find deadlocks which we describe later. Even if these analyses target deadlocks arising out of incorrect condition variable usage, they still focus on a subset of those kinds of deadlocks. They cannot predict a wide variety of deadlocks that we can with CheckMate. For example, they cannot predict deadlocks involving waits with two locks and the deadlock in Example 5.4.

There has been some previous work on finding typestate errors. 2ndStrike [45] predicts typestate errors and automatically confirms them by controlling the thread scheduler. It requires typestate specifications to be provided, but PRETEX automatically infers the specifications from observed executions. But, the technique of controlling the thread scheduler to confirm real typestate errors is similar to DeadlockFuzzer in which we confirm resource deadlocks. Other techniques [97] have been proposed to predict violations of safety properties in multithreaded programs which can be expressed using temporal logic. Temporal logic might not be sufficient to express many typestate properties, and therefore these techniques will not be able to check multithreaded programs against those typestate properties.

There has been work on automatically confirming potential concurrency bugs. Random testing techniques have been proposed [33, 103] that introduce noise (using `yield`, `sleep`, `wait` (with timeout)) to a program execution to increase the possibility of the exhibition of a synchronization bug. Although these techniques have successfully detected bugs in many programs, they have a limitation. These techniques are not systematic as the primitives `sleep()`, `yield()`, `priority()` can only advise the scheduler to make a thread switch, but cannot force a thread switch. As such they cannot pause a thread as long as required to create a real bug.

More recently, a few techniques have been proposed to confirm potential bugs in concurrent programs using random testing. Havelund et al. [11] use a directed scheduler to confirm that a potential deadlock cycle could lead to a real deadlock. However, they assume that the thread and object identifiers do not change across executions. Similarly, ConTest [85] uses the idea of introducing noise to increase the probability of the occurrence of a deadlock. It records potential deadlocks using a Goodlock algorithm. To check whether a potential deadlock can actually occur, it introduces noise during program execution to increase the probability of exhibition of the deadlock. DeadlockFuzzer differs from ConTest in the following ways. ConTest uses only locations in the program to identify locks. We use context information and object abstractions to identify the run-time threads and locks involved in the deadlocks; therefore, our abstractions give more precise information about run-time objects. Moreover, we explicitly control the thread scheduler to create the potential deadlocks, instead of adding timing noise to program execution. DeadlockFuzzer, being explicit in controlling scheduler and in identifying objects across executions, found real deadlocks in large benchmarks with high probability.

DeadlockFuzzer is part of the *active testing framework* [65] that provides support for designing dynamic analyses for concurrency bugs, and for designing checkers to confirm bugs by actively controlling the thread scheduler. RaceFuzzer [96] uses the framework to find real races with high probability, and AtomFuzzer [87] to find atomicity violations. RaceFuzzer only uses statement locations to identify races and does not use object abstraction or context information to increase the probability of race detection. As shown in Section 9.4, simple location information is not good enough for creating real deadlocks with high probability.

10.2 Static analysis and Model checking

Static analyses have also been developed to find different concurrency issues. These analyses find errors without executing the program, but often report many false positives (i.e. reports that do not correspond to real errors). There has been a lot of work [79, 42, 8, 35, 107, 112, 84] on statically finding resource deadlocks. Most of this work deals with implicitly or explicitly computing the lock graph of a program, and finding cycles in it. Static techniques often give no false negatives, but they often report many false positives. For example, the static deadlock detector developed by Williams et al. [112] reports 100,000 deadlocks in Sun's JDK 1.4 [1], out of which only 7 are real deadlocks. Type and annotation based techniques [16, 42] help to avoid deadlocks during coding, but they impose the burden of annotation on programmers.

There has been some work on statically finding communication deadlocks. Hovemeyer and Pugh [60] present several common deadlock patterns in Java programs that are checked by their static tool FindBugs, including many involving condition variables such as unconditional wait, wait with more than one lock held, etc. Their patterns can miss out on other kinds of communication deadlocks like missed notifications and the deadlocks we had seen earlier in Figure 5.4 in Chapter 5. Von Praun [107] also statically detects waits that may execute with more than one lock held, and waits that may be invoked on locks on which there is no invocation of a notify. His approach too cannot detect missed notifications and the deadlock in Figure 5.4.

There has been a lot of work [104, 30, 38, 72] on finding typestate errors statically in sequential programs. There is not much work for multithreaded programs. Yang et. al. [116] have proposed a static technique to combine typestate checking with concurrency analysis. The approach approximates thread interactions statically, and can result in high false positive rates.

Model checking [59, 49, 31, 54, 20] systematically explores all thread schedules to find concurrency bugs in them. Since it explores all possible schedules, it can find bugs that can occur only in specific and rare schedules. However, model checking fails to scale to large multithreaded programs due to the exponential increase in the number of thread schedules with execution length.

[1] They reduce the number of reports to 70 after applying various unsound heuristics

Part II

Testing Distributed Systems

Chapter 11

Introduction

Large-scale distributed systems are increasingly in use in today's cloud era. For example, there are large-scale file systems (Hadoop File System [100]) and databases (Cassandra [75]), computing platforms (Amazon EC2 [2]), and software as a service systems (Google Apps) that run on clusters with thousands of commodity machines to serve hundreds of thousands of simultaneous clients. Distributed systems of such large-scale have an enormous amount of interaction going on between the commodity machines or nodes to carry out their computation and to have consistent views regarding the system state. Given how popular and pervasive cloud systems have become, and how they affect millions of users everyday, it is essential that we test the systems thoroughly before deploying them. But, the complexity and non-determinism in the interaction between different nodes, and even within a single node in large-scale distributed systems result in such an enormous number of feasible executions that it is hard to test them all with reasonable computing resources and time.

In addition to testing the correctness of cloud systems, we also need to test the robustness of the systems against diverse hardware failures such as machine crashes, disk errors, and network failures. The commodity machines that these systems are built out of are not fully reliable and can exhibit frequent failures [29, 55, 89, 94, 106]. Even if the software on cloud systems is built with reliability and failure tolerance as primary goals [25, 29, 46], the recovery protocols in it are often buggy. For example, the developers of Hadoop File System [100] have dealt with 91 recovery issues over its four years of development [50]. There are two main reasons for the presence of recovery issues. Sometimes developers fail to anticipate the kinds of failures that a system can face in a real environment (*e.g.*, only anticipate fail-stop failures like crashes, but forget to deal with silent failures like data corruption), or they incorrectly design or implement the failure recovery code. There have been many serious consequences (*e.g.*, data loss, unavailability) of recovery bugs in real cloud systems [18, 21, 22, 50]. Hence, not only do we need to test the functional correctness of a system, we also need to test its robustness against failures as failures are not the exception but rather the rule for large-scale systems.

Model checking verifies the correctness of a distributed system by exhaustively checking all possible executions of the system. Different executions resolve differently the non-

determinism in the ordering of events (i.e. operations) that are of interest to the tester (e.g. sending and receiving of messages, and timeouts). Thus, model checking verifies that each possible ordering of events results in correct system behavior. Model checking is effective in that it can discover subtle corner-case bugs that can occur only when some events execute in a specific order. But, since it exhaustively checks all possible executions, it does not scale well to large systems that can have millions of feasible executions. Similarly, to test the robustness of a system against failures, we can emulate (or "inject") all possible kinds and combinations of failures during execution and test the recovery of the system from those failures. Again, this process of exhaustively checking against all failures is effective in finding corner-case recovery issues, but this does not scale well to huge failure spaces with hundreds of thousands of possible failures and failure combinations that can occur during execution. In fact, the state-of-the-art of testing against failures employs exhaustive testing only for single failures during execution, and tests against multiple-failure combinations by randomly choosing some combinations and testing them. Injecting multiple failures is important because failing a system when it is already in the process of recovering from previous failures can often reveal recovery bugs that cannot be found by injecting a single failure. Random testing of multiple-failure combinations can be effective in discovering bugs, but it can miss serious corner-case bugs that get triggered only when specific failures occur at specific points of execution.

There have been different techniques proposed in the past that deal with improving the efficiency and scalability of model checking. Partial-order reduction [48, 40] is an effective way to reduce the number of executions that model checking has to explore. It essentially finds which events are "independent" (that is, events that result in the same system state irrespective of the order in which they execute), and explores only a single ordering of such events. This reduces the state space of executions, but the reduction still might not be sufficient to scale model checking to large systems. Partial-order reduction uses the happens-before relation [82] between events that is computed by observing which shared resources do the events access and how are they synchronized with respect to each other to decide which events are dependent and which are not. However, even if two events might be dependent according to the happens-before relation, they might not be dependent for the tester's objectives as either order of two events results in the same degree of fulfillment of the tester's objectives (*e.g.*, high source coverage). Testing any one of the two orders of the events would suffice for the tester, but both the orders would be exercised by partial order reduction. Similarly, when testing for failures, a tester might consider two different failures or failure combinations to be equivalent for her testing objectives and might want to test just one of them instead of exploring them both. In fact, we have found from our personal experience and from talking to developers of cloud systems that often a tester has a good idea about how to prune down the large space of failures so that the failures that are explored enable her to achieve her testing objectives. For example, a tester might want to fail only one representative subset of nodes in a system, or explore only a subset of all possible failure types, or explore failures that occur at different source locations to obtain a high source code coverage. Thus, if we can enable testers to easily express their intentions or intuitions to

the testing process, then the testing process can use those intentions or intuitions to prune down large state space of executions or failures in such a way that exploring the pruned down spaces help achieve the testers' objectives.

In this second part of the thesis, we present work that builds tools and frameworks to enable testers to easily express their intuitions and knowledge to direct the testing process towards those executions or failures that are more likely to help achieve the testers' objectives. We provide the right abstractions of events, failures, and system executions that can be used by testers to express their intuitions and knowledge without having any knowledge about the internals of the testing process. Testers can express their intuitions and intentions in Python which is a popular scripting language. We first show how we can build a customizable (or programmable) testing tool that tests the functional correctness of distributed systems (in the absence of failures), and then we present another programmable tool for failure testing. One common challenge that arises for both the tools is understanding what a tester might want to express, and figuring out the right kinds of abstractions to provide to the tester that would help her to express that and the right kind of tool architecture that would help to compute and expose those abstractions. We explain policies (expression of tester's intuition or knowledge), abstractions, and tool architecture in detail in the subsequent chapters.

In the subsequent chapters, we first provide an overview of how the programmable tools work with policies (Chapter 12), and then explain the background definitions (Chapter 13) that we use to describe the tools in Chapters 14 and 15. We describe how we have implemented our tools in Chapter 16, and evaluate various aspects of the tools in Chapter 17. Finally, we compare the tools with related work in Chapter 18.

Chapter 12

Overview

In this chapter, we illustrate using two examples how testers can use their knowledge and intuition to guide the testing process towards interesting executions without having any knowledge about the internals of the testing process. The first example is a simplified version of the leader election protocol, and the second example is about hardware failures during I/O operations in a distributed file system.

12.1 Example 1: Leader Election

In leader election, a group of nodes (or processes) in a distributed system exchange messages (or votes) amongst each other to decide on a node that will function as the leader. The leader then assumes various responsibilities like attending to clients' requests and facilitating coordination among different nodes. When leader election starts, each node considers itself to be the leader and sends a message to each of the other nodes voting for itself. For example, Node 1 sends "V = 1" (V stands for "Vote") to the other nodes to indicate that it is voting for itself. Similarly, Node 2 sends "V = 2", Node 3 sends "V = 3", and so on. A node also updates its current view of the leader when it receives a higher vote from another node. For example, when Node 1 receives a vote "V = 2" from Node 2, it updates its current view of the leader to Node 2 from Node 1, and sends votes "V = 2" to the other nodes to inform them of this change. This process is repeated until a majority of nodes agree on one particular node as the leader.

Figure 12.1 illustrates the possible orderings of messages at a node (Node 3) during the initial stage of leader election in a system with five nodes. Initially, Node 3 is going to consider itself as the leader, but is going to receive four votes, one each from each of the other nodes proposing itself as the leader. Since the votes arrive concurrently, there is no deterministic order in which they are received and processed. For example, orderings #1 and #2 in Figure 12.1 show two different orders of votes from Node 4 and Node 5, and orderings #3 and #4 show different orders of votes from Node 1 and Node 2. Different orders of messages might trigger different system behaviors (e.g. orderings #1 and #2 result

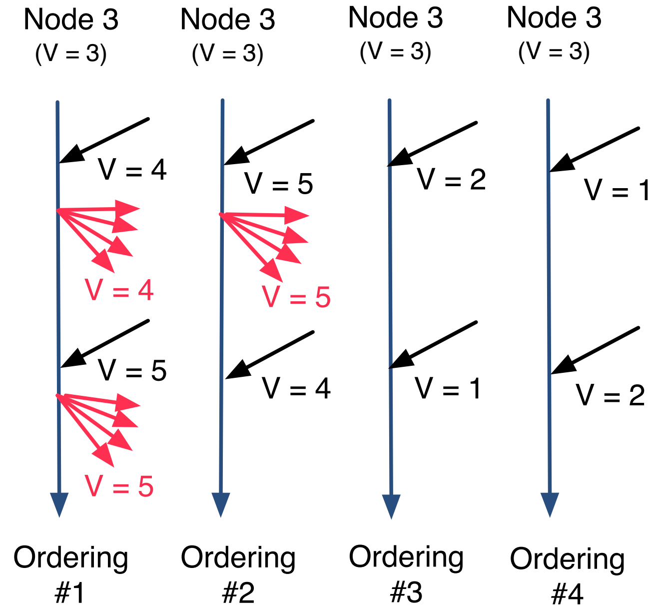


Figure 12.1: Different possible orderings of messages at Node 3 in the initial stage of the leader election protocol. V stands for "Vote".

in different votes being broadcast by Node 3), and thus, it is necessary that we test the correctness of system behavior under different message orders.

If we want to test the correctness of leader election under all possible message orderings, then we can naïvely order the four incoming votes at Node 3 in 24 (= 4!) different ways with each ordering being in conjunction with orderings of messages at other nodes. Even with partial-order reduction [40] (that can potentially reduce the number of executions to be explored by considering only a single ordering of "independent" operations or messages), we would have to do 24 orderings since all messages at a receiver node are received through the same socket and also queued onto the same queue and hence are considered to be dependent by the happens-before relation in the partial-order reduction. Sending of messages at a node might be considered to be independent by partial-order reduction [101], but receiving of messages are considered to be dependent if they access the same shared resources (*e.g.*, sockets, queues) at the receiver node.

From our knowledge of leader election, we might know that not all vote orderings result in distinct system behaviors at Node 3. Exercising any one of such orderings would thus suffice for us. Similarly, there might be votes that we might know would result in different behaviors if ordered differently. We would like to exercise all possible orderings of such votes. We show examples of both scenarios next and also explain how we can exploit our knowledge of the election protocol to direct the model checking process towards the orderings that are interesting to us.

```
def dep (v1, v2, S):
        bool b1 = (v1.receiver == v2.receiver)
        bool b2 = (v1.vote != v2.vote)
        bool b3 = (v1.vote > curVote(v1.receiver))
        bool b4 = (v2.vote > curVote(v2.receiver))
        return (b1 and b2 and b3 and b4)
```

Figure 12.2: Reorder votes v_1 and v_2 only if they are different and greater than the current leader at the receiver node

Consider Ordering #3 and Ordering #4 again in Figure 12.1. Since the votes from Node 1 and Node 2 do not affect the current leader at Node 3, exercising either order of the two votes would suffice. We do not want to explore both orders of the votes as we know that the different orders are not going to affect the current leader at Node 3 differently (in fact, they are not going to affect the current leader at Node 3 at all). Now consider Ordering #1 and Ordering #2 in the same figure. When Node 3 receives "V = 4" in Ordering #1, it updates its current leader to "Node 4" from "Node 3" and broadcasts this change to the other nodes by sending votes for Node 4 to them. When it later receives "V = 5", it again updates its current leader to "Node 5" and informs other nodes of this change by sending votes for Node 5. In Ordering #2, Node 3 receives "V = 5" first, and thus updates its current leader directly to "Node 5". Later when it receives "V = 4", the vote has no effect on the receiver's current leader as the vote is lower than the current leader. Thus, receiving "V = 4" before "V = 5" impacts the current leader differently than receiving "V = 5" before "V = 4". Therefore, we want to explicitly order both the votes to test that the system operates correctly under both the orders.

To sum up the search policy that we want to adopt when exploring different vote orderings: we want to explore both orders of two votes at a receiver node iff the votes are higher than the current leader at that node and are different. Figure 12.2 presents how we can express the policy in our programmable tool to direct the testing process to explore only the orderings that we want. If we want to explore both orders of two given votes, then the policy returns true; otherwise false. With this policy, we would explore only 2 orderings of the four votes at Node 3; one in which "V = 4" occurs before "V = 5", and the other in which "V = 5" is before "V = 4". Thus, we get a speed-up of 12 (= 24/2) over naïve or exhaustive exploration (model checking) or even with partial-order reduction in the number of executions explored. For real-world systems and workloads and for ordering all messages across all nodes, the speed-ups can be much higher.

12.2 Example 2: Distributed File System

Consider the code segment in Figure 12.3 that runs on a distributed file system with two nodes, A and B. When a client reads from or writes to a file, there are various disk and

```
         Node A                        Node B
A1. write(B, msg);            B1. write(A, msg);
A2. read(B, header);          B2. read(A, header);
A3. read(B, body);            B3. read(A, body);
A4. write(B, msg);            B4. write(A, msg);
A5. write(Disk, buf);         B5. read(Disk, buf);
```

Figure 12.3: I/Os executing in two nodes in a distributed file system

network reads and writes that would execute in each node. The very simple distributed program in the figure illustrates how I/Os execute in each node. The I/Os can fail in different ways during execution. For example, there might be a transient hardware failure during the execution of an I/O, or a node might crash before an I/O, or the network might fail resulting in the failure of a network read or write, or the data read might be corrupted. Thus, we should test the robustness of distributed systems against different kinds of failures that they can face. The state-of-the-art way to test against failures is to inject (or emulate) failures during execution and then to check if the system could correctly recover from the injected failures or not. For example, we can inject a transient I/O failure at the write call on line A1 by executing code that throws an IOException instead of executing the write call.

Let us suppose that a tester wants to test against crashes before `read` and `write` calls, and that she wants to inject two crashes in an execution. One possible combination is to crash before the write at A4 and then to crash before the write at B5. Overall, since there are 5 possible points to inject a crash on every node, there are $5^2 * N(N-1)$ possible ways to inject two crashes, where N is the number of participating nodes (N = 2 in the above example). Again, considering many other factors such as different failure types, and more failures that can be injected during recovery, the number of all possible failure sequences can be too many to exhaustively explore with reasonable computing resources and time. Moreover, increasing the number of failures to inject per execution can potentially result in a combinatorial explosion in the number of failure sequences that can be tested. Thus, multiple and diverse failures can pose a serious challenge for testing robustness of distributed systems.

Instead of trying to automatically prune down the failure space (set of failure sequences), we let the testers specify which failure sequences are interesting to them or which are equivalent to them so that only one sequence out of a group of equivalent sequences would be exercised. For example, let us say that the tester for the example in Figure 12.3 wants to only test against node crashes before disk I/Os. Thus, the only pair of crashes that she would like to test is a crash before the disk write at line A5 and a crash before the disk read at line B5. The tester can write the policy in Figure 12.4 to direct the failure testing process to only exercise the crashes at A5 and B5. Note that the tester does not require a detailed understanding of the internals of the failure testing process to be able to write policies. Our tool provides high-level abstractions of execution points where failures can

```
def flt (fs):
   for f in fs:
      fp = FIP(f)
      isCrash = (fp['failure'] == 'crash')
      isBefore = (fp['place'] == 'before')
      isDisk = (fp['target'] == 'disk')
      if not (isCrash and isBefore and isDisk):
             return False
   return True
```

Figure 12.4: Inject crashes before disk I/Os

occur (FIP in Figure 12.4 which is explained later in Chapter 13), and history of previous executions to enable testers to easily express a variety of suitable policies according to their testing objectives and budget.

12.2.1 Examples of failure space pruning

To give an idea about the kinds of specific failure sequences a tester might want to focus on, we present some examples of testing situations and the kinds of policies that a tester might want to write in those situations from our personal experience and our conversation with developers of cloud systems.

Failing a component subset: Let's suppose a tester wants to test a distributed write protocol that writes four replicas to four machines, and let's suppose that the tester wants to inject two crashes in all possible ways during execution to show that the protocol can survive and continue writing to the two surviving machines even after the crashes. A brute-force technique will inject failures on all possible combinations of two nodes (i.e., $\binom{4}{2}$). However, to do this quickly, the tester might wish to specify a policy that just injects failures in *any* two nodes.

Failing a subset of failure types: Another way to prune down a large failure space is to focus on a subset of the possible failure types. For example, let's imagine a testing process that, at every disk I/O, can inject a machine crash or a disk I/O failure. Furthermore, let's say the tester knows that the system is designed as a crash-only software [19], that is, all I/O failures are supposed to translate to system crash (followed by a reboot) in order to simplify the recovery mechanism. In this environment, the tester might want to just inject I/O failures but not crashes because it is useless to inject additional crashes as I/O failures will lead to crashes anyway. Another good example is the rack-aware data placement protocol common in many cloud systems to ensure high availability [43, 100]. The protocol should ensure that file replicas should be placed on multiple racks such that if one rack goes down, the file can be accessed from other racks. In this scenario, if the tester wants to test the rack-awareness property of the protocol, only rack failures need to be injected (*e.g.*, vs.

single node or disk failures).

Coverage-based policies: A tester might want to speed up the testing process with some coverage-based policies. For example, let's imagine two different I/Os (A and B) that if failed could initiate the same recovery path that performs another two I/Os (M and N). To test recovery, a tester should inject more failures in the recovery path. A brute-force method will explore 4 executions by injecting two failures at AM, AN, BM, and BN (M and N cannot be exercised by themselves unless A or B has been failed). But, a tester might wish to finish the testing process when she has satisfied some code coverage policy, for example, by stopping after all I/O failures in the recovery path (at M and N) have been exercised. With this policy, she only needs to explore 2 executions with failures at AM and AN.

Domain-specific optimization: In some cases, system-specific knowledge can be used to reduce the number of failures. For example, consider 10 consecutive Java read I/Os that read from the same input file (*e.g.*, `f.readInt()`, `f.readLong()`, ...). In this scenario, disk failure can start to happen at any of these 10 calls. In a brute-force manner, a tester would run ten experiments where disk failure begins at 10 different calls. However, with operating system's knowledge, the tester might inject disk failure only on the first read. The reasoning behind this is that a file is typically already buffered by the operating system after the first call. Thus, it is unlikely (though possible) to have earlier reads succeed and the subsequent reads fail. In our experience (Chapter 17), by reducing the number of such individual failures, we can greatly reduce the number of multiple failure combinations that we have to test.

Failing probabilistically: Multiple failures can also be reduced by only exercising them if the likelihood of their occurrence is greater than a predefined threshold [91, 105]. This is useful especially if the tester is interested in correlated failures. For example, two machines put within the same rack are more likely to fail together compared to those put across in different racks [43]. A tester can use real-world statistical data to implement policies that are based on failure probability distributions.

In the subsequent chapters, we explain our programmable tools for model checking and failure testing of distributed systems. We show how we architect the tools so that they can provide simplified high-level abstractions to testers that the testers can then use to write a variety of search or pruning policies.

Chapter 13

Background Definitions

In this chapter, we introduce the definitions that we later use to describe the programmable tools. A programmable tool should be able to provide the right abstractions of the relevant aspects of system execution so that they can be used by testers to write appropriate search or pruning policies. In order to provide the right abstractions, we need to identify the information that a tester might want when writing a policy. We show next how we identify the interesting context and abstract it and expose it to testers in our programmable tools.

13.1 Execution related abstractions

Let us suppose that we have identified the statements that are relevant to the tester. For example, for testing system correctness under different network message orders, the statements that send and receive network messages are relevant, and for failure testing, statements executing I/Os are relevant. We abstract out the execution of a relevant statement (also called an *event*) as a map from a set of keys $\mathcal{K}$ to a set of values $\mathcal{V}$. A key k in $\mathcal{K}$ represents a part of the static or dynamic context associated with the event. For example, k could be 'sender' to denote the node that sent a message, or 'receiver' to denote the node that received the message, or 'type' to denote the type of the message, or 'packet' to denote the contents of the message, or 'stack' to denote the call stack at the node before or after the event. Table 13.1 gives an example of the abstraction of an event that we might encounter while testing correctness under different message orders. The abstraction encapsulates the context of a message sent by Node 2 to Node 3. A tester can obtain the context that she wants regarding an event by accessing the relevant keys of the event abstraction. We provide a variety of keys to enable testers to access different kinds of context. Identifying the relevant statements and the context to provide in abstractions solves the problem of deciding what context information to expose to testers.

Table 13.2 gives an example of an abstraction of an event we might encounter during failure testing. We call the abstraction of an event where a failure can occur as a failure-injection point (`fip`). A key k in a `fip` represents a part of the static or dynamic context

Key	**Value**
sender	Node 2
receiver	Node 3
type	Election
packet	Vote = 2
...	...

Table 13.1: Event abstraction

`fip` B5

Key	**Value**
func	read()
loc	Read.java (line L5)
node	B
target	file f
stack	⟨stack trace⟩
...	...

Possible Failures at `fip` B5	`fit`
Crash	(crash, B5) / $\texttt{B5}_c$
Corruption	(corruption, B5) / $\texttt{B5}_{cr}$
Disk failure	(disk failure, B5) / $\texttt{B5}_d$

Table 13.2: Failure-Injection Point (`fip`) and Failure-Injection Task (`fit`)

associated with the event as before. For example, k could be 'func' to represent the function call being executed. It would be mapped to the name of the function in the `fip`. Other examples for k are: 'loc' for the location of the function call in the source code, 'node' for the node ID on which the execution occurs, and 'target' for the target of the I/O executed by the function call (*e.g.*, the name of the file being written to in case of a disk write I/O). The left-hand side of Table 13.2 shows the `fip` (labeled B5) corresponding to the execution point at the `read` call at line L5 in node B. We denote the set of all failure-injection points by $\mathcal{P}$.

We call a pair of a failure type (*e.g.*, crash, disk failure) and a failure-injection point as a failure-injection task (`fit`). Thus, a `fit` $f \in \mathcal{F} \times \mathcal{P}$, where $\mathcal{F}$ denotes the set of all failure types. Given a failure-injection point, there are different types of failures that can be injected at that point. For example, the right-hand side of Table 13.2 shows different `fit`s that can be formed for the `fip` illustrated in the same table for three different types of failures (crash, data corruption, and disk failure). Exercising a `fit` $f = (ft, fp)$ means injecting the failure type ft at the `fip` fp. Since, in failure testing, we are interested in injecting multiple failures during execution in addition to single failures, we also consider sequences of failure-injection tasks. We denote the set of all sequences of failure-injection tasks by $\mathcal{Q}$. We call a sequence of failure-injection tasks as a *failure sequence* in short.

13.2 Profiling executions

We also profile executions and record them so that testers can use the profiles of past experiments while writing their policies. For example, for failure testing, we profile an execution by the set of failure-injection points observed in that execution. Failure-injection points are typically built out of I/O calls, library calls, or system calls, and these calls can be used to approximately represent an execution of the system under test. Thus, an execution profile $exp \in 2^{\mathcal{P}}$.

Let `allFips`: $\mathcal{Q} \rightarrow 2^{\mathcal{P}}$ and `postInjectionFips`: $\mathcal{Q} \rightarrow 2^{\mathcal{P}}$ be the functions that return execution profiles in failure testing. Given a failure sequence *fs*, `allFips`(*fs*) returns the execution profile consisting of all `fip`s observed during the experiment in which *fs* is injected, and `postInjectionFips`(*fs*) returns the set of all `fip`s observed *after fs* has been injected. For the empty sequence (), `allFips` and `postInjectionFips` both return the set of all `fip`s seen in the execution in which no failure is injected.

In order to obtain the relevant execution and profiling information, we need to instrument the system to keep track of the execution of relevant statements. For example, for testing correctness under different message orders, we should instrument network message sends and receives, and for failure testing, we should instrument the I/O statements (*e.g.*, disk reads and writes, network reads and writes, etc.). For message orders, we also need to instrument the statements that assemble the network message contents so that we can expose the contents in the abstractions. Thus, after we have identified the relevant statements and context, we need to instrument those statements to build abstractions and profiles during execution.

In the next two chapters, we explain the internals of our programmable tools, and also demonstrate how event abstractions are used by testers to write appropriate policies to drive the testing processes in the tools.

Chapter 14

Programmable Model Checking

In this chapter, we explain our programmable tool PCheck to model check (that is, systematically and exhaustively test) distributed systems. PCheck has three components: the event generator (MC Facilitator) that executes the given system, and identifies and abstracts out the relevant events during execution, the policy framework (MC Driver) that lets a tester express the event orderings that she wants to test, and the model checker (MC Engine) that systematically explores the space of event orderings that are described in the policy written by the tester. The MC Facilitator provides suitable abstractions for events that can be used by a tester to write a variety of policies. In this work, we have focused on testing system protocols that involve significant messaging between nodes. Thus, message sends and receives are the events that MC Facilitator tracks and exposes. For testing other kinds of protocols and properties, we can design the MC Facilitator accordingly to expose events relevant to those protocols and properties. In the following sections, we explain how the three components of PCheck interact, and also explain each component in detail.

14.1 Overall Architecture

Figure 14.1 shows the three components of PCheck, and their mutual interactions. Given a system $\mathcal{X}$, the MC Facilitator executes $\mathcal{X}$, and keeps executing it repeatedly until the MC Engine declares the exploration of event orderings to be over. During each execution, the Facilitator identifies the relevant events (*e.g.*, execution of a message send or receive), and blocks their execution so that a previously unexplored ordering can be imposed on the execution of those events. When the system execution reaches a stable state where no new relevant event is observed, the MC Facilitator sends abstractions of the blocked events to the MC Engine. The MC Engine decides which of the blocked events to execute next. The next unblocked event might generate more relevant events, and thus, the MC Facilitator waits for the system execution to again reach a stable state where no new event is generated. It then repeats the process of sending the blocked events to the MC Engine that decides the next event to be executed. This process is repeated until the system $\mathcal{X}$ finishes its execution.

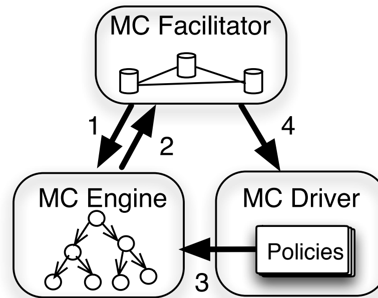


Figure 14.1: Architecture of PCheck. **1**: Blocked events, **2**: Model checking decision, **3**: Policies, and **4**: Abstractions

The MC Engine uses the history of past system executions that it has seen, and the policies written by the tester to decide which event ordering it wants to explore in the current execution. For example, say the tester's policies deem two events A and B to be dependent, that is, the policies express the tester's intention to test both orders of A and B. Then, if event A has already been executed before event B in a previous execution at a specific system state, then the MC Engine might explore B before A when the current execution reaches that system state. Thus, if both A and B have been blocked, then it would ask the MC Facilitator to execute B next in order to explore the ordering of B before A. The MC Engine uses tester's intuitions expressed in policies to decide which events are dependent rather than using the happens-before relation that is used by traditional partial-order reduction to determine dependent events.

The MC Driver provides a framework using the event abstractions provided by the MC Facilitator to enable a tester to write appropriate policies. A tester can use her knowledge and intuition to express which events should not be reordered by the MC Engine even if they are considered dependent by partial-order reduction since the difference in the system states that the different orders of the events result in does not matter to the tester. For example, consider two events A and B that are considered dependent by partial-order reduction. Both orders of the events would thus be exercised by partial-order reduction. But, let us assume that the set of statements that would be covered if we execute event A and then event B is same as the set of statements that would be covered if we execute event B and then event A. Thus, if source code coverage is the testing objective, then we need not explore both A after B and B after A – either of the two orders would suffice even if the two orders would result in different system states. The tester can express this knowledge in a policy using the event abstractions provided by the MC Facilitator. Using the policy, the MC Engine would explore only one of the two orders of A and B.

Algorithm 12 `Execute(`$\mathcal{S}$`)`

1: **Input:** `Stable system state` $\mathcal{S}$ `to begin execution from`
2: **while** `not hasTerminated(`$\mathcal{S}$`)` **do**
3: `NE := nextEvents(`$\mathcal{S}$`)`
4: e `:= nextEventToExecute(NE)`
5: $\mathcal{S}$ `:= nextStableState(`$\mathcal{S}$`,` e`)`
6: **end while**

14.2 MC Facilitator

The MC Facilitator is responsible for executing a system and identifying the relevant events during execution. Since we are interested in verifying messaging intensive protocols, the relevant events for us are network message sends and receives. We want to verify that the system executes correctly even when the order in which messages are received and executed at various nodes of the system is changed. For other protocols, we can re-design the Facilitator to track and expose events relevant to those protocols. MC Facilitator identifies the statements that are relevant to a messaging protocol, and instruments them so that it can track their execution and generate appropriate abstractions (Chapter 13) when they execute. Abstractions are used by the MC Engine to decide the next event to execute, and also by testers to write policies.

Algorithm 12 illustrates how the MC Facilitator executes the system $\mathcal{X}$ under test. Given $\mathcal{X}$, the Facilitator executes it until it reaches a stable state $\mathcal{S}$ (Lines 1 and 5). A *stable state* is an execution state in which no new relevant event is going to be observed even if we execute $\mathcal{X}$ further. For example, consider $\mathcal{X}$ to be consisting of three nodes each of which sends a message to the other two. After we have executed $\mathcal{X}$ and observed and blocked all the six message receives, we would have reached a stable state since no more message receive can be generated unless one of the blocked receives is unblocked and executed. Thus, at a stable state, we have to choose and execute a blocked relevant event in order for $\mathcal{X}$ to make progress in the protocol under consideration. Note that there might be non-determinism in the way $\mathcal{X}$ executes until it reaches a stable state, but we are not controlling that non-determinism. We are only controlling the non-determinism in the ordering of events relevant to the protocol (*e.g.*, leader election protocol) that we are considering, and exploring different possible ways in which those events can execute. We tolerate the non-determinism during execution as the system transitions from one stable state to the next, and thus, save ourselves the cost of performing a full-fledged model checking of the system. We do a targeted model checking, that is, exploring all orders of only the relevant events and not all events during execution. It is possible that we might fail to explore an event ordering since we do not control all sources of non-determinism. In such cases of failure, we resolve the event ordering in the execution randomly and then explore orderings around the random event ordering in subsequent executions.

After $\mathcal{X}$ has reached a stable state $\mathcal{S}$, we abstract out the relevant events `NE` that have been blocked and can be executed next (`nextEvents(`$\mathcal{S}$`)`), and send the abstractions

to the MC Engine. The MC Engine uses the history of executions that it has seen so far and the tester-written policies to decide which of the blocked events to execute next (`nextEventToExecute(NE)` in Algorithm 15). The MC Facilitator unblocks and executes that event e, and waits until $\mathcal{X}$ reaches stable state again (`nextStableState(`$\mathcal{S}$`, `e`)`).

Determining if a system has reached stable state varies from system to system and protocol to protocol. The ideal way to determine this would be for the tester to provide a predicate on the system state that evaluates to true if a stable state has been reached. But, since writing such a predicate would require in-depth knowledge of the implementation details of $\mathcal{X}$ and internals of the protocol, we use a heuristic to determine the stable state. If the set of blocked events remains constant after a configurable amount of time (say, 1 second), then we assume we have reached a stable state. If the tester has a good understanding of the system and the protocol, then she can of course override the heuristic by providing a more precise predicate to determine stable state.

MC Facilitator repeats the process of querying the MC Engine for the next event to execute at the next stable state and proceeds as before. At each stable state $\mathcal{S}$, the Facilitator also checks to see if the protocol being tested has already finished its execution at that state (`hasTerminated(`$\mathcal{S}$`)`). For example, if we are testing the leader election protocol, then we would execute the system $\mathcal{X}$ until a leader is elected. The tester provides the predicate `hasTerminated` to help the Facilitator determine when to consider a system execution to be over. The MC Facilitator keeps executing a system repeatedly until the MC Engine determines that all possible event orderings that are interesting to the tester have been explored. After finishing an execution, the Facilitator queries the Engine via `doNext?()` (Algorithm 16) to decide whether to execute the system again.

14.3 MC Driver

The MC Driver provides a policy framework in which testers can easily express which events they want to reorder. They can use their intuitions and knowledge to understand and express which events are dependent, that is, which events can potentially lead to two different states when ordered differently such that the difference in the two states can affect the degree of fulfillment of the testing objectives. For example, in the leader election protocol, the message receive events at a node that contain different proposals for the leader are dependent since the receiver node's view of the leader might potentially be different for different orderings of the two events.

Let $\mathcal{E}$ denote the set of all possible events. Then, the predicate `dep` : $\mathcal{E} \times \mathcal{E} \times \mathcal{E}^* \rightarrow$ Boolean holds for the tuple (e_i, e_j, `S`) if the events e_i and e_j are considered to be dependent by the tester in the system state that results after executing the event sequence `S`. The tester can implement her policy of deciding event dependence by suitably implementing `dep` using the abstractions and libraries provided by the MC Driver. The predicate `dep` is used by the MC Engine (Algorithm 14) to compute the event orderings that it should explore.

```
λ e1, e2, S . (
        let r1 = (e1['receiver']) in
        let r2 = (e2['receiver']) in
        let sameReceiver = (r1 == r2) in
        let t1 = (e1['type']) in
        let t2 = (e2['type']) in
        let isFAck1 = (t1 == 'FINALACK') in
        let isFAck2 = (t2 == 'FINALACK') in
                sameReceiver ∧ isFAck1 ∧ isFAck2
)
```

Figure 14.2: Gossip protocol: Reorder message receive events e_1 and e_2 that are final acks for gossip requests

```
λ e1, e2, S . (
        let r1 = e1['receiver'] in
        let r2 = e2['receiver'] in
        let sameReceiver = (r1 == r2) in
        let t1 = e1['type'] in
        let t2 = e2['type'] in
        let isAck1 = (t1 == 'ACK') in
        let isAck2 = (t2 == 'ACK') in
        let wr1 = (e1['wrID']) in
        let wr2 = (e2['wrID']) in
        let diffWrs = (wr1 ≠ wr2) in
                sameReceiver ∧ isAck1 ∧ isAck2 ∧ diffWrs
)
```

Figure 14.3: Client write protocol: Reorder message receive events e_1 and e_2 that are acks for different write requests

Figure 14.2 presents an example of the `dep` predicate. Let us consider a system that has multiple nodes with each node contacting a small subset of other nodes asking them to provide it with gossip information regarding the live nodes in the system that they are aware of. The communication between two nodes involves a number of messages with the final message being a final ack sent by a node to the node that requested gossip information. Instead of reordering all messages that are exchanged between all the nodes, using the policy in Figure 14.2 we test all orderings of only the final acks received at the nodes. This is because the accumulated gossip information (gossip state) at a node is updated only when a final ack is received from another node regarding a new piece of gossip information. Thus, two different orders of final acks can potentially result in two different gossip states at a node. Other messages cannot update the gossip state, and hence their different orderings cannot result in different gossip states at a node.

Figure 14.3 gives another example implementation of `dep`. Let us say that we want to

```
λ e1, e2, S . (
        let r1 = e1['receiver']) in
        let r2 = e2['receiver']) in
        let v1 = e1['packet']) in
        let v2 = e2['packet']) in
        let sameReceiver = (r1 == r2) in
        let diffVotes = (v1 ≠ v2) in
        let cv = curVote(r1, S) in
        let isGreaterV1 = isGreater(v1, cv) in
        let isGreaterV2 = isGreater(v2, cv) in
                sameReceiver ∧ diffVotes ∧ isGreaterV1 ∧ isGreaterV2
)
```

Figure 14.4: Leader election protocol: Reorder message receive events e_1 and e_2 that have votes that are greater than the current leader at the receiver

test multiple clients writing concurrently to the same file in a distributed file system. The servers in the file system redirect client write requests to the elected leader in the system. For each write request, the leader asks for votes or acks from all servers in the system. If the leader obtains enough acks for a write request, it commits that request. Instead of testing all possible orderings of the messages sent between the clients, the leader, and the servers in the system, we can use the policy (that is, the predicate) in Figure 14.3 to test only the different orderings of the acks received at the leader. For different orders in which acks are received at the leader, the sequence in which the write requests are committed might be different and thus, the state of the file being written to might be different. Reordering the messages that are sent by the leader to other nodes asking for acks cannot directly impact the order in which the leader commits the write requests and changes the state of the file. Thus, we do not reorder those messages sent by the leader.

Figure 14.4 presents yet another example implementation of the `dep` predicate for the leader election protocol. This example illustrates how we can use the event sequence `S` provided as a parameter to decide whether two events are dependent in the system state that we reach after executing `S`. The predicate orders two incoming election votes at a node if the values of the votes are different and if each vote is greater than the current view of the leader at that node. That is, if we have two message receive events at the same node, and the votes in the two messages `v1` and `v2` are different and greater than the current view of the leader `cv` at that node, then we test both orders of the two messages. If either of the votes is less than `cv`, then that vote is not going to affect `cv`. Thus, we reorder only when the incoming votes are greater than `cv`. We use the sequence `S` to compute what the current leader is for the receiver node. Also, two equal votes in different messages are going to affect `cv` in the same way irrespective of the order in which they are processed. Therefore, we reorder two incoming messages only if their votes are different.

Algorithm 13 (PROG-DPOR(frontier, N))

```
1: if frontier == EmptyStack then
2:     return
3: end if
4: for each (Q, k) in (top(frontier) \ done) do
5:     add Q to done
6:     update(frontier, Q)
7:     NE := Enabled(Q)
8:     if NE ≠ ∅ then
9:         e := choose(NE)
10:        if k ≥ 0 then
11:            push(frontier, {(Q.e, k)})
12:        end if
13:        PROG-DPOR(frontier, N)
14:    end if
15: end for
16: pop(frontier)
```

Algorithm 14 update(frontier, S)

```
1: l := |S|
2: e_last := S_l
3: for i in (l-1) to 1 do
4:     e_prev := S_i
5:     sq := S_1...(i-1)
6:     if dep(e_last, e_prev, sq) ∧ (e_prev ∈ Enabled(sq)) ∧ (e_last ∈ Enabled(sq)) ∧ (∃ k > 0 (sq,
   k) ∈ frontier_(i-1)) then
7:         add (sq.e_last.e_prev, k - 1) to frontier_(i+1)
8:     end if
9: end for
```

14.4 MC Engine

The MC Engine records information regarding past executions observed and the current execution to decide which event orderings to explore in the current execution. We restrict the number of reorderings that we can do in a particular execution with a tester-provided parameter N. If N is 1, then the MC Engine would explore all possible executions that reorder or switch at most one pair of events starting with the original system execution. Similarly, if N is 2, then we explore all executions that we can reach starting from the original execution by switching the order of at most two pairs of events, and so on. Algorithms 13, 14, 15, and 16 explain how the MC Engine explores the space of executions with at most N event reorderings. The MC Engine essentially performs a dynamic partial-order reduction [41] (Algorithm 13) on the execution tree of feasible event orderings. The reduction in [41] would consider all executions that can be reached starting from the original system execution by

Algorithm 15 `nextEventToExecute(NE)`

```
1: Input: Set of abstract blocked events NE
2: if |S| < |Q| then
3:     e := Q_{|S|+1}
4: else
5:     add S to done
6:     update(frontier, S)
7:     e := choose(NE)
8:     if k ≥ 0 then
9:         push(frontier, {(S.e, k)})
10:    end if
11: end if
12: S := S.e
13: return e
```

Algorithm 16 `doNext?()`

```
1: add S to done
2: update(frontier, S)
3: while (top(frontier) \ done) == ∅ do
4:     pop(frontier)
5: end while
6: if frontier == EmptyStack then
7:     return false
8: end if
9: (Q, k) := choose(top(frontier) \ done)
10: S := EmptySequence
11: return true
```

performing any number of event reorderings, but we restrict ourselves to executions that can be reached with N reorderings so that testers can fix and increase N based on the time and resources that they have.

The MC Engine implements the partial-order reduction using Algorithms 15 and 16. Instead of using the happens-before relation to decide whether two events are dependent and can result in different system states when reordered [41], we let a tester decide when to consider two events to be dependent (Algorithm 14). Often two events might be judged to be dependent using the happens-before relation (say, since they both access the same shared memory location), but a tester might consider them to be not dependent since the changes in the system state for the two different orders of the events do not affect the tester's objectives (*e.g.*, code coverage). Thus, using tester's intuition and knowledge, we might reduce the space of event orderings more than what we can using only traditional partial-order reduction.

The MC Engine implements a depth-first search of the execution tree using dynamic partial-order reduction (Algorithm 13). The execution tree represents feasible event orderings

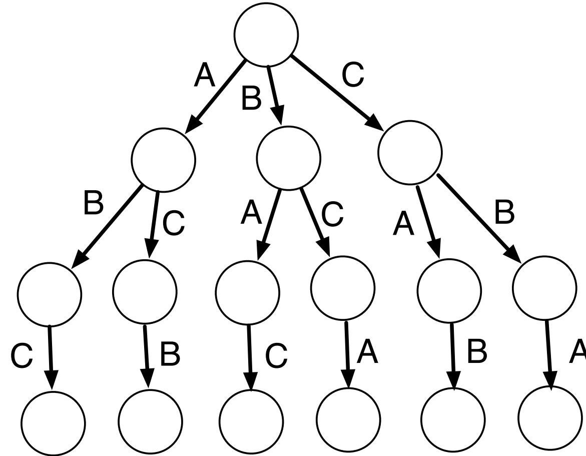


Figure 14.5: An example execution tree. The MC Engine performs depth-first search with dynamic partial-order reduction on execution trees.

(or sequences) that can occur during execution. Figure 14.5 gives an example of an execution tree. The root node represents the empty event sequence. We have an outgoing edge e from a node if event e is enabled after the event sequence leading to the node has been executed. For example, in Figure 14.5, there are three events initially that are enabled: A, B, and C. If we choose A and execute it, then we have B and C that are enabled after the execution of A. It is possible that new events get generated after executing A, but we do not consider that situation to keep the example simple.

For depth-first search, we maintain a stack `frontier` that keeps track of the nodes that can be explored next, and a set `done` that keeps track of the nodes that have been explored. $\texttt{frontier}_i$ ($i \geq 0$) represents the i-th record of the stack with a higher i denoting a record higher up in the stack or closer to the top of the stack. Initially, `frontier` has only one element $\{$(`EmptySequence`, `N`)$\}$ where `EmptySequence` is the empty event sequence and `N` is the tester-provided parameter that restricts the number of event reorderings in a single execution. A stack element (`Q`, k) specifies the space of event orderings that begin with the sequence `Q` and that have at most k event reorderings after `Q` has been executed. We move forward along a path in the search tree starting from the root by arbitrarily picking an enabled event e at each node (`choose` in Algorithm 13) until we reach the end of the system execution. In the process of moving along the path, we add backtrack nodes into `frontier` (Algorithm 14) that we later go back to and explore. For example, assume that N is 1 and that we choose A, B, and C in order as we move forward in the tree in Figure 14.5. After moving along A, we push $\{$(`A`, 1)$\}$ to `frontier`. Similarly, we push $\{$(`AB`, 1)$\}$ after B, and $\{$(`ABC`, 1)$\}$ after C. We also add backtrack nodes as we move forward that we explain next.

Algorithm 14 explains how we add backtrack nodes into `frontier` when exploring the node n that we reach after the event sequence `S`. For the last event $e_{\texttt{last}}$ leading into n, we

go backwards along the path (or the sequence S) that reaches n to find an event $e_{\texttt{prev}}$ with which we can reorder $e_{\texttt{last}}$. The events $e_{\texttt{prev}}$ and $e_{\texttt{last}}$ are enabled together during execution and are considered to be dependent by the tester. The tester implements the function `dep` in MC Driver (Section 14.3) that takes two events and an event sequence as arguments and decides whether the events are dependent or not at the state the system is in after the event sequence is executed. Furthermore, the sequence of events before $e_{\texttt{prev}}$ (`sq`) contains less than N (N - k with $k > 0$) event reorderings. Thus, reordering $e_{\texttt{prev}}$ and $e_{\texttt{last}}$ after `sq` would not exceed N. We add the backtrack node `sq`.e_{last}.e_{prev} (sequence `sq` followed by e_{last} and then e_{prev}) to the stack and also decrement k to denote that we can perform 1 less than k reorderings in the rest of the execution after the sequence `sq`.e_{last}.e_{prev} since we have added one more event reordering after `sq`. We find all events $e_{\texttt{prev}}$ with which $e_{\texttt{last}}$ can be reordered, and add appropriate backtrack points for them.

Consider the example in Figure 14.5 again. After we have executed A and B, we go backwards to find an event that can be reordered with B. We find that the only other event A can be reordered with B since they are both enabled at the same at the beginning of the execution. Let us assume that the tester has also expressed that she wants to test both orders of A and B in her policies. Thus, we add (BA, 0) to $\texttt{frontier}_2$ changing $\texttt{frontier}_2$ to {(AB, 1), (BA, 0)}. We decrease the number of reorderings for BA to 0 since we are already using up our budget of 1 to reorder A and B, and hence, we cannot perform any more reorderings after BA. After executing C after A and B, we again go backwards from C to find events with which we can reorder C. We find that C can be reordered with both B and A, and thus, add backtrack nodes (ACB, 0) and (CA, 0) to $\texttt{frontier}_3$ and $\texttt{frontier}_2$ respectively. After executing ABC, we pick ACB from frontier to explore next. After executing ACB, we add backtrack nodes as before and proceed to the next execution. In the end, we would have executed ABC, ACB, BAC, CAB, CBA, and BCA.

The MC Engine interacts with the MC Facilitator and implements the depth-first search with partial-order reduction using Algorithms 15 and 16. In the algorithms, S is the current event sequence explored at any point of time, and Q is the sequence that we want to execute first in the next execution. Q is initially the empty sequence. The number of reorderings that can be done in the remaining execution is denoted by k which is initially set to N (maximum number of reorderings allowed). In Algorithm 15, the MC Engine moves forward along a path by arbitrarily choosing an enabled event at each node, and adds backtrack nodes along the way. The set of enabled events NE at a node is provided by the MC Facilitator. When backtracking to a node Q, the algorithm first replays the sequence Q before arbitrarily choosing nodes and moving forward in the sub-tree rooted at Q. Algorithm 16 decides the next backtrack point to explore. If there is no backtrack point that we have not yet explored, it declares the search over and indicates to MC Facilitator that it does not have to execute the system again. If it finds a backtrack node Q, it instructs the Facilitator to execute the system again. Algorithm 15 then guides the execution to first follow the sequence Q of events and then to arbitrarily choose enabled events till completion.

Thus, we have seen how testers can use policies to reduce the number of executions explored during model checking. In the next chapter, we explain how testers can use policies

to reduce the number of failure sequences exercised during failure testing.

Chapter 15

Programmable Failure Injection

This chapter presents our programmable failure testing tool PreFail [63]. PreFail has two components: the FI engine and the FI driver as shown in Figure 15.1 (FI stands for failure-injection). The FI engine is the component that injects failures in the system under test, and the FI driver is the component that takes tester-specified policies to decide where to inject failures. The FI engine exposes failure related abstractions to the FI driver that can be used by the testers in their policies. In the following sections, we first illustrate the test workflow in PreFail (§15.1), and then we explain the FI engine (§15.2), the FI driver and policies (§15.3), and finally the detailed algorithm of the test workflow (§15.4).

15.1 Test Workflow

Figure 15.2 shows an example scenario of the testing process in PreFail. The tester specifies three failures as the maximum number of failures to inject in an execution of the system under test. The FI engine first runs the system with zero failure during execution (i.e. without injecting any failure during execution). During this execution, it obtains the set of all execution points where failures can be injected (*i.e.*, *failure-injection points* as described in Chapter 13): `A`, `B`, and `C`. Let us assume that we are interested only in crashes, and let A_c, B_c, and C_c denote the injection of crashes (*i.e.*, *failure-injection tasks* as described in Chapter 13) at the failure-injection points `A`, `B`, and `C`, respectively (for ease of reading, a failure-injection task X_c is represented as `X` in a box in Figure 15.2). Using the tester-specified policies, suppose PreFail prunes down the set of failure-injection tasks to A_c and B_c, and then exercises each failure-injection task in the pruned down set.

After exercising a failure-injection task, the FI engine records all failure-injection points seen where further crashes can be injected. For example, after exercising A_c (that is, injecting a crash at `A`), the FI engine observes the failure-injection points `D` and `E`. A letter in a circle in Figure 15.2 represents a failure-injection point observed in an experiment. From information regarding observed failure-injection points, the FI engine creates the set of sequences of two failure-injection tasks A_cD_c and A_cE_c that can be exercised while injecting two crashes in

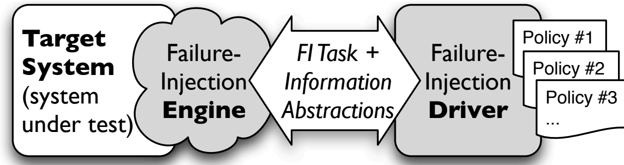


Figure 15.1: PreFail's Architecture. The figure shows the separation of PreFail into failure-injection engine and driver.

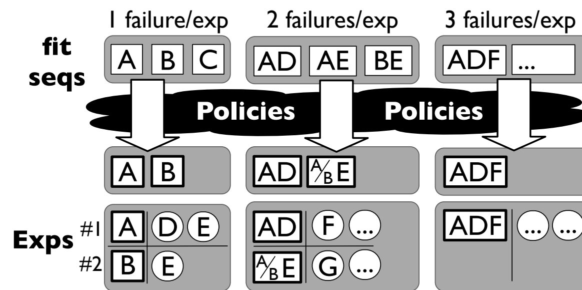


Figure 15.2: PreFail's Test Workflow

an execution. Similarly, it creates $\mathtt{B}_c\mathtt{E}_c$ after observing the failure-injection point E in the execution that exercises $\mathtt{B}_c$.

As mentioned before, the number of all sequences of failure-injection tasks that can be exercised tends to be large. Thus, PreFail again uses the tester-specified policies to reduce this number. For example, a tester might want to test just one sequence of two crashes that exercises $\mathtt{E}_c$ as the second crash. Thus, PreFail would automatically exercise just one of $\mathtt{A}_c\mathtt{E}_c$ and $\mathtt{B}_c\mathtt{E}_c$ to satisfy this policy instead of exercising both of them. The step from injecting two failures to three failures per execution is similar.

15.2 FI Engine

The failure-injection tasks described above are created by the FI engine. The FI engine interposes different execution points in the system under test and injects failures at those points. The target failure-injection points and the range of failures that can be injected all depend on the objective of the tester. For example, interposition can be done at Java/C library calls [50, 77], TCP-level I/Os [27], disk-level I/Os [90], POSIX system calls [74], OS-driver interfaces [62], and at many other points. Depending on the target failure-injection points, the range of failures that can be injected varies. In our work, we interpose all I/O related to calls to Java libraries and emulate hardware failures by supporting diverse failure types such as crashes, disk failures, and network partitioning at node and rack levels.

Algorithm 17 `fpGen`

```
1: Inputs: A filter predicate flt and a set of failure sequences FS
2: Output: A set of failure sequences FS_P
3: FS_P = {}
4: for fs in FS do
5:     if flt(fs) then
6:         FS_P = FS_P ∪ {fs}
7:     end if
8: end for
9: return FS_P
```

Algorithm 18 `cpGen`

```
1: Inputs: A cluster predicate cls and a set of failure sequences FS
2: Output: A set of failure sequences FS_P
3: FS_P = {}
4: E = FS/R_cls
5: for e in E do
6:     fs = select an element from e randomly
7:     FS_P = FS_P ∪ {fs}
8: end for
9: return FS_P
```

The FI driver tells the FI engine to run a set of experiments that satisfy the written policies. An *experiment* is an execution of the system under test with a particular failure scenario (could be one or multiple failures). For example, the FI driver could tell the FI engine to run one experiment with one specific failure (*e.g.*, a crash before a specific write) or two failures (*e.g.*, two simultaneous crashes before two specific writes in two nodes).

15.3 FI Driver

Based on the abstractions explained earlier in Chapter 13, the FI driver provides support for writing predicates that it uses to generate policies that express how to prune the failure space. For convenience and brevity, whenever we say that a tester writes a policy, we mean that the tester writes the predicate that is later used by the FI driver to generate the policy. A policy is a function $p : 2^{\mathcal{Q}} \rightarrow 2^{\mathcal{Q}}$, where $\mathcal{Q}$ is the set of all failure sequences as described in Chapter 13. It takes a set of failure sequences, and returns a subset of the sequences to be explored by the FI engine. Testers can use the failure and execution point abstractions, and execution profiles provided by the FI engine in their predicates. There are two different kinds of predicates that can be written to generate two different kinds of policies: *filter* and *cluster* policies. PreFail can also compose the policies generated from different predicates to obtain more complex policies.

```
λ fs . (
      let ((ft₁, fp₁), ..., (ftₙ, fpₙ)) = fs in
      let isCrash(ft) = (ft == crash) in
      let inSetup(fp) = fp['stack'] has 'setup' in
                    ⋀           isCrash(ftᵢ) ∧ inSetup(fpᵢ)
               i∈{1,...,n}
)
```

Figure 15.3: Setup-stage filter: Return true if all `fits` $(ft_1, fp_1), \ldots, (ft_n, fp_n)$ in *fs* correspond to a crash within the `setup` function.

15.3.1 Filter Policy

A filter policy uses a tester-written predicate `flt`: $\mathcal{Q} \rightarrow$ `Boolean`. The predicate takes a failure sequence *fs* as an argument and implements a condition that decides whether to exercise *fs* or not. Algorithm 17 explains how a filter policy works. Given a predicate `flt`, the function `fpGen` : $(\mathcal{Q} \rightarrow \texttt{Boolean}) \rightarrow (2^{\mathcal{Q}} \rightarrow 2^{\mathcal{Q}})$ (implemented in PreFail) generates a filter policy out of it. The policy takes a set of failure sequences *FS*, applies the `flt` predicate on each sequence *fs*, and retains *fs* in its result set FS_P if the predicate holds for it.

15.3.2 Cluster Policy

A cluster policy uses a tester-implemented predicate `cls`: $\mathcal{Q} \times \mathcal{Q} \rightarrow$ `Boolean`. The predicate takes two failure sequences as arguments, and returns true if the tester considers them to be similar (*e.g.*, exercising either of them would result in the same test coverage), and false otherwise. The predicate implicitly implements an equivalence relation $\mathrm{R}_{\texttt{cls}} = \{(fs_1, fs_2) \mid \texttt{cls}(fs_1, fs_2)\}$. Algorithm 18 shows how a cluster policy works. Given a `cls` predicate, the function `cpGen` : $(\mathcal{Q} \times \mathcal{Q} \rightarrow \texttt{Boolean}) \rightarrow (2^{\mathcal{Q}} \rightarrow 2^{\mathcal{Q}})$ (implemented in PreFail) generates a cluster policy out of it. The policy uses the predicate to partition its argument set of failure sequences *FS* into disjoint subsets $FS/\mathrm{R}_{\texttt{cls}}$. It then randomly selects one failure sequence *fs* from each equivalence class. Thus, the tester implements her notion of equivalence of failure sequences, and the policy uses the equivalence relation to select failure sequences such that all equivalence classes in its argument set of failure sequences are covered.

15.3.3 Example Policies

We give brief examples of how one can use the filter and cluster policies. Suppose that a tester is interested in testing the tolerance of the setup stage of a distributed systems protocol against crashes. The tester can write the `flt` predicate in Figure 15.3. The filter policy `fpGen`(`flt`) would retain a failure sequence *fs* only if every `fit` in *fs* corresponds to a crash in the setup stage (execution of the `setup` function).

```
λ fs1, fs2. (
        let rec(fs) = allFips(fs) \ allFips(()) in
        let eq(fs1, fs2) = (rec(fs1) == rec(fs2)) in
        let (f11, ..., f1m) = fs1 in
        let fs1P = (f11, ..., f1(m-1)) in
        let (f21, ..., f2n) = fs2 in
        let fs2P = (f21, ..., f2(n-1)) in
                (eq(fs1P, fs2P) ∧ (f1m == f2n) ∧ (m ≥ 2)
                        ∧ (n ≥ 2))
)
```

Figure 15.4: Recovery path cluster: Cluster two failure sequences if their last `fit`s are the same and their prefixes (that exclude the last `fit`s) result in the same recovery path.

In failure testing, since we are concerned with testing the correctness of *recovery* paths of a system, one way to reduce the number of failure sequences to test would be to cluster them according to the recovery paths that they would lead to. Out of all failure sequences that would lead to a particular recovery path, we can just choose and test one. To achieve this, we can write the cluster predicate in Figure 15.4. If two failure sequences fs_1 and fs_2 have the same last `fit`, and their prefixes that leave the last `fit` out (fs_{1P} and fs_{2P} respectively) result in the same recovery path, then we can consider fs_1 and fs_2 to be equivalent in terms of the recovery paths that they would lead to since they involve injecting the same failure at the same execution point in the same recovery path. PreFail's test workflow is such that when deciding whether to test a failure sequence (*e.g.*, fs_1), all of its prefixal sequences (*e.g.*, fs_{1P}) would have already been tested, and thus we would have already seen the recovery paths that they lead to. Figure 15.4 uses the function `rec` to characterize a recovery path. It uses the set of all `fip`s seen in the recovery path to characterize it. From all `fip`s observed during an execution in which a failure sequence is injected, we subtract out the `fip`s that are observed during normal program execution (that is, when no failure is injected) to obtain the `fip`s seen in the recovery path. More details about recovery path clustering can be found in Section 15.5.4.

PreFail also enables composition of policies. For example, the policies that use the predicates in Figures 15.3 and 15.4 (`fpGen(flt)` and `cpGen(cls)`) can be composed to first filter out those sequences that have crashes in the setup stage, and then to cluster the filtered sequences according to the recovery paths that they would lead to. Section 15.5 shows how to write the policies in Python in PreFail, and also gives many other examples of policies.

15.4 Test Workflow Algorithm

Having outlined the major components of PreFail, this section presents the detailed algorithm of PreFail's test workflow (Algorithm 19). PreFail takes a system `Sys` to test, a list of tester-written predicates `Preds`, and the maximum number of failures N to inject in

Algorithm 19 PreFailTest Workflow

1: `INPUT: System under test (Sys), List of flt and cls predicates (Preds),`
   `Maximum number of failures per execution (`N`)`
2: $FS_c = \{()\}$
3: $FS_n = \{\}$
4: **for** $0 \le i \le N$ **do**
5:     **for** `each failure sequence` fs `in` FS_c **do**
6:         `Execute Sys and inject` fs `during execution`
7:         `Profile execution using fips observed during execution`
8:         **for** `each fit` f `computed from a fip in postInjectionFips(`fs`)` **do**
9:             $fs' =$ `Append` f `to` fs
10:             $FS_n = FS_n \cup fs'$
11:         **end for**
12:     **end for**
13:     $FS_n =$ `Prune`(`Preds`, FS_n)
14:     $FS_c = FS_n$
15:     $FS_n = \{\}$
16: **end for**

Algorithm 20 `Prune`(`Preds`, FS)

1: $FS_P = FS$
2: **for** `predicate pr in Preds` **do**
3:     **if** `pr is a filter predicate` **then**
4:         `p = fpGen(pr)`
5:     **end if**
6:     **if** `pr is a cluster predicate` **then**
7:         `p = cpGen(pr)`
8:     **end if**
9:     $FS_P =$ `p`(FS_P)
10: **end for**
11: `return` FS_P

an execution of the system. The testing process runs in $N+1$ steps. At step i ($0 \le i \le N$), the FI engine of PreFail executes the system `Sys` once for each failure sequence of length i that it wants to test, and injects the failure sequence during the execution of the system. FS_c is the set of all failure sequences that should be tested in the current step, and FS_n is the set of failure sequences that should be tested in the next step. Initially FS_c is set to a singleton set with the empty failure sequence as the only element. Therefore, in step 0 the FI engine executes `Sys` and injects an empty sequence of failures, i.e. it does not inject any failure. The FI engine observes the `fips` that are seen during execution, computes `fits` from them, and adds singleton failure sequences with these `fits` to FS_n. Therefore, FS_n has failure sequences that the FI engine can exercise in the next step, i.e. in the $i = 1$ step. Before PreFail proceeds to the next step, it prunes down the set FS_n using the predicates written

by testers. The predicates in `Preds` are used to generate policies that are then applied to FS_n (Algorithm 20). The policy generated from the first predicate is applied first to FS_n, the second policy is then applied to the result of the first policy and so on. Note that the order of predicates is important since the policies generated from them may not commute. In step $i = 1$, FS_c is set to the pruned down FS_n from the previous step, and FS_n is reset to the empty set. For each failure sequence *fs* in FS_c, the failure-injection tool executes `Sys` and injects *fs* during execution. For each `fit` f that it computes from a `fip` observed after *fs* has been injected (that is, a `fip` in `postInjectionFips`(*fs*)), it generates a new failure sequence *fs*′ by appending f to *fs*, and adds *fs*′ to FS_n. After `Sys` has been executed once for each failure sequence in FS_c, PreFail prunes down the set FS_n with predicates and moves to the next step. This process is repeated till the last step.

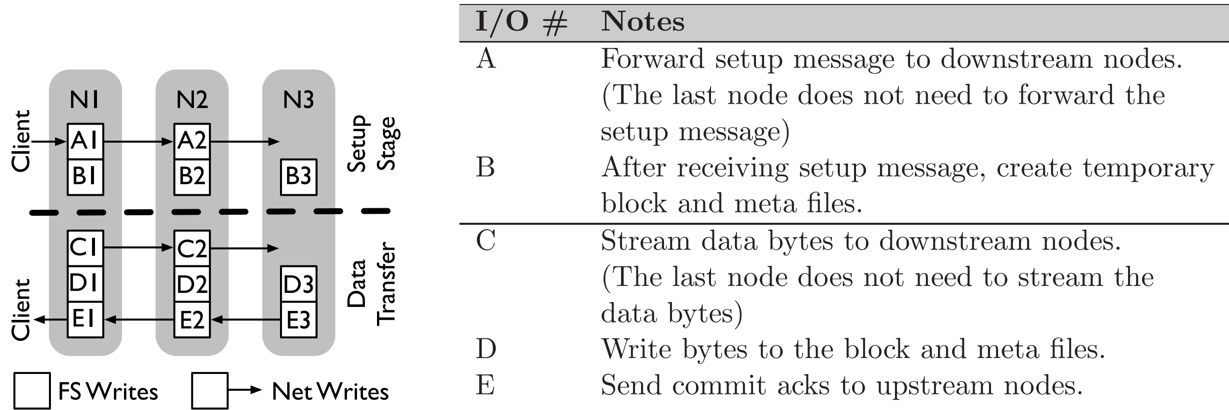
15.5 Crafting Pruning Policies

In this section, we present the pruning policies that we have written and their advantages. More specifically, we present our integration of PreFail to Hadoop File System (HDFS) [100], an underlying storage system for Hadoop MapReduce [4], and show the policies that we wrote for it. We begin with an introduction to HDFS and then present the policies.

Overall, we make three major points in this section. First, by clearly separating the failure-injection mechanism and policy and by providing useful abstractions, we can write many different pruning policies clearly and concisely. Second, we show that policies can be easily composed together to achieve different testing objectives. Finally, we show that some policies can be reused for different target systems. We believe these advantages show the power of PreFail. We chose Python as the language in which testers can write policies in PreFail, though any other language could have also been chosen.

15.5.1 HDFS Primer

HDFS is a distributed file system that can scale to thousands of nodes. Here we describe the HDFS write protocol in detail. Figure 15.5 shows a simplified illustration of the write I/Os (both file system and network writes) occurring within the protocol. Each box represents an I/O, and thus a failure-injection point. For ease of reading, we label each failure-injection point with an alphabetical symbol plus the node ID. The protocol by default stores three replicas in three nodes. The client forms a pipeline (Client-N1-N2-N3) with the three nodes where replicas of a file will be stored. The client obtains these target nodes from the master (communication between client and master is not shown in the figure). The write protocol is divided mainly into two stages: the setup stage and the data transfer stage; we will later see how the recovery for each stage is different. For simplicity, we have not shown many other I/Os such as other acknowledgment and disk I/Os. We also have not shown the rack-aware placement of replicas.



I/O #	Notes
A	Forward setup message to downstream nodes. (The last node does not need to forward the setup message)
B	After receiving setup message, create temporary block and meta files.
C	Stream data bytes to downstream nodes. (The last node does not need to stream the data bytes)
D	Write bytes to the block and meta files.
E	Send commit acks to upstream nodes.

Figure 15.5: HDFS Write Protocol

```
1 def cls (fs1, fs2):
2  rs1 = abstractOut(fs1, 'node')
3  rs2 = abstractOut(fs2, 'node')
4  return (rs1 == rs2)
```

Figure 15.6: Ignore nodes cluster: Return true if two failure sequences have the same failures with the same contexts not considering the nodes in which they occur.

Our FI engine is able to emulate hardware failures on every I/O (every box in Figure 15.5). As illustrated, there are 13 failure points that the FI engine interposes in this write protocol. (Note that, in reality, the write protocol performs more than 40 I/Os). At every I/O, the FI engine can inject a crash, a disk failure (if it's a disk I/O), or a network failure (if it's a network I/O). The figure also depicts many possible ways in which multiple failures can occur. For example, two crashes can happen simultaneously at failure-injection points `B1` and `B2`, or a disk failure at `D1` and a network failure at `E3`, and many more. Interested readers can learn more about HDFS from [100, 111] and from our extended technical report (which depicts the write protocol in more detail) [64].

15.5.2 Pruning by Failing a Component Subset

In distributed systems like HDFS, it is common to have multiple nodes participating in a distributed protocol. As mentioned earlier, let's say we have N participating nodes, and the developer wants to inject two failures on two nodes. Then there are $\binom{N}{2}$ failure sequences that one could inject. Worse, on every node (as depicted in Figure 15.5), there could be many possible points to exercise the failure on that node.

To reduce the number of failure sequences to test, a developer might just wish to inject

```
1 def flt (fs):
2  last = FIP (fs [ len(fs) - 1 ])
3  return not explored (last, 'loc')
```

Figure 15.7: New source location filter: Return true if the source location of the last `fip` has not been explored

```
1 def cls (fs1, fs2):
2  last1 = FIP (fs1[len(fs1) - 1])
3  last2 = FIP (fs2[len(fs2) - 1])
4  return (last1['loc'] == last2['loc'])
```

Figure 15.8: Source location cluster: Return true if the `fips` in the last `fits` have the same source location.

failures at all possible failure-injection points in *any* two nodes. She can write a cluster policy that uses the function in Figure 15.6 to cluster failure sequences that have the same context when the node is not considered as part of the context. With this policy, the developer can direct the FI engine to exercise failure sequences with two failures such that if the FI engine has already explored failures on a pair of nodes then it should not explore the same failures on a different pair of nodes. The function `abstractOut` in the figure removes the mappings for nodes from the `fips` in its argument failure sequence. Using Figure 15.5 as an example, a failure sequence with simultaneous crashes at D1 and D2 is equivalent to another with crashes at D2 and D3.

We also want to emphasize that this type of pruning policy could be used for other systems. Consider a RAID system [88] with N disks that a tester wishes to test by injecting failures at any two of its N disks. To do this, we definitely need a FI engine that works for RAID systems, but we can re-use much of the policy that we wrote for distributed systems for RAID systems. The only difference would be in the keys in the `fips` whose mappings we want to remove (*i.e.*, for distributed systems we removed the mappings for the 'node' key in Figure 15.6, for RAID systems we remove the mappings for 'disk' key).

15.5.3 Pruning via Code-Coverage Objectives

Developers can achieve high-level testing objectives using policies. One common objective in the world of testing is to have some notion of "high coverage". In the case of failure testing, we can write policies that achieve different types of coverage. For example, a developer might want to achieve a high coverage of source locations of I/O calls where failures can happen.

To achieve high code-coverage with as few experiments as possible, the tester can simply compose the policies that use the `flt` function shown in Figure 15.7 and the `cls` function shown in Figure 15.8. The filter policy explores failures at previously unexplored source loca-

fit	Recovery Path (Fig. 15.9)	SL	SL+N
$A1_c$	{ABCDE}×{234}	□	□
$B2_c$	{ABCDE}×{134}	□	△
$C1_c$	{FGI}×{23}, {CDE}×{23}	■	■
$C2_c$	{FGJ}×{13}, {CDE}×{13}	●	●
$D1_c$	{FG}×{23}, {CDE}×{23}	○	○
$E2_c$	{FG}×{13}, {CDE}×{13}	○	▽

Table 15.1: HDFS Write Recovery

tions by filtering out a failure sequence if the `fip` in its latest `fit` (`last`) has an unexplored source location. The function `FIP` in Figure 15.7 returns the `fip` in the argument `fit`. The function `explored` returns true if a failure has already been injected at the source location in the last `fip` in a previous failure-injection experiment. For brevity, we do not show the source code of these functions. The `cls` function in Figure 15.8 clusters failure sequences that have the same source location in their last `fits`. Thus, after the filter policy has filtered out failure sequences that have unexplored source locations, the cluster policy would cluster the failure sequences with the same unexplored source location into one group. With these policies, PreFail would exercise a failure sequence for each unexplored source location.

15.5.4 Pruning via Recovery-Coverage Objectives

In failure testing, since we are concerned with testing the correctness of *recovery* paths of a system, another useful testing goal is to rapidly explore failures that lead to different recovery paths. To do this, a tester can write a cluster policy that clusters failure sequences leading to the same recovery path into a single class. PreFail can then use this policy to exercise a failure sequence from each cluster, and thus exercise a different recovery path with each failure sequence. Below, we first describe the HDFS write recovery protocol, and then explain the whole process of recovery-coverage based pruning in two steps: characterizing recovery path, and clustering failure sequences based on the recovery characterization.

HDFS Write Recovery

As mentioned before, the HDFS write protocol is divided mainly into two stages: the setup stage and the data transfer stage. Table 15.1 shows in detail the recovery I/Os for the two stages, that is, the I/Os that occur during execution while recovering from an injected failure (or failure sequence). The first column shows the `fits`. $A1_c$ is the `fit` for crash at the I/O A1. For simplicity, we do not distinguish between an I/O and the failure-injection point that corresponds to the execution of the I/O. The second column shows the recovery paths returned by the `getRecoveryPath` function (Figure 15.9) for every `fit` shown in the first column[1]. For brevity, we use ×; {AB}×{12} represents the I/Os A1, A2, B1, and B2. The third and fourth columns represent two ways of characterizing the recovery path; the same

shape represents the same class of recovery path. For example, the third column represents the characterization shown in Figure 15.10 which uses source location (SL) to characterize recovery. The fourth column uses source location and node ID (SL+N) to characterize recovery. We will gradually discuss the contents of the table in the following sections.

The recovery for each stage in the HDFS write protocol is different. In the setup stage, if a node crashes, the recovery protocol will repeat the whole write process again with a new pipeline. For example, in the first row of Table 15.1, after `N1` crashes at I/O A1 ($A1_c$), the protocol executes the entire set of I/Os again (ABCDE) in the new pipeline (`N2-N3-N4`). However, if a node crashes in the second stage, the recovery protocol will only repeat the second stage with some extra recovery I/Os on the surviving datanodes. For example, in the fifth row of Table 15.1, after `N1` crashes at D1 ($D1_c$), the protocol first performs some synchronization I/Os (FG), and then repeats the second stage I/Os (CDE) on the surviving nodes (`N2` and `N3`).

Characterizing Recovery Path

```
1  def getRecoveryPath (fs):
2   a = allFips(fs)
3   a0 = allFips([])
4   rPath = a - a0
5   return rPath
```

Figure 15.9: Obtaining Recovery Path FIPs

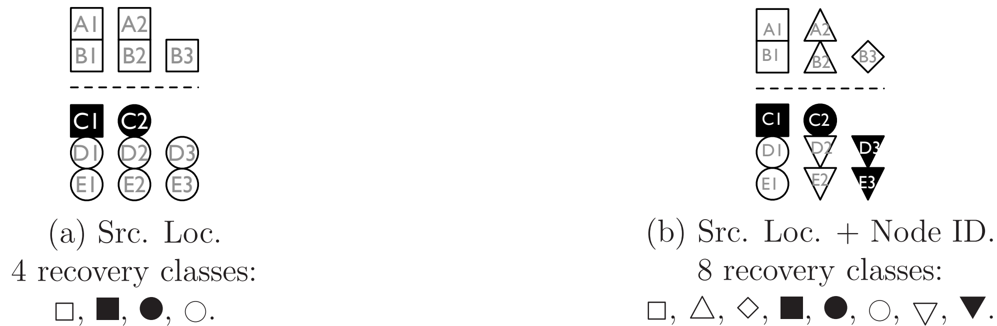
To write a recovery clustering policy, a tester has to first decide how to characterize the recovery path taken by a system. One way to characterize would be to use the set of `fips` observed in the recovery path. Figure 15.9 returns the "difference" of the `fip`s observed in the execution in which a failure sequence `fs` is injected and the `fips` in the execution in which no failure is injected. The difference can be thought of as the `fips` that are observed in the "extra" execution that results or the recovery path that is taken when the failure sequence is injected. Line 2 in Figure 15.9 uses the function `allFips` (Chapter 13) to get the set of all `fips`, `a`, observed during the execution in which `fs` is injected. Line 3 obtains the set of `fip`s observed when no failure is injected (represented by "[]"). Line 4 performs the "diff" of the two sets to obtain the `fip`s in the recovery path taken when `fs` is injected.

A tester can use the set of `fip`s observed in the recovery path to characterize the recovery path. Thus, two failure sequences that result in the same set of `fips` in the recovery path are

[1]The reader might wonder why the I/Os A, B, C, D, and E appear again in the recovery paths even though the `getRecoveryPath` function returns the "diff" between the I/Os in the execution with failures and in the normal execution path, and thus should exclude those I/Os. The answer is that these I/Os are executed in the recovery path too, but with different contexts (e.g. different message content, different generation number) that we incorporate in the `fip`. For simplicity, we do not discuss these detailed contexts here.

```
1 def eqvBySrcLoc (fs1, fs2):
2     r1 = getRecoveryPath (fs1)
3     r2 = getRecoveryPath (fs2)
4     c1 = abstractIn(r1, 'loc')
5     c2 = abstractIn(r2, 'loc')
6     return c1 == c2
```

Figure 15.10: Considering only source locations to judge equivalence of recovery paths



(a) Src. Loc.
4 recovery classes:
□, ■, ●, ○.

(b) Src. Loc. + Node ID.
8 recovery classes:
□, △, ◇, ■, ●, ○, ▽, ▼.

Figure 15.11: Recovery Classes of HDFS Write Protocol

considered to be equivalent. Instead of using all of the context in the `fips`, the tester might abstract out the `fips` and use only part of the context in them to characterize a recovery path. For example, the tester might want to use only the source locations of `fips`. Thus, she might consider two recovery paths to be the same if the I/Os in them occur at the same set of source locations. The function in Figure 15.10 considers this relaxed characterization of recovery paths. It returns true if two failure sequences result in the system executing I/Os at the same set of source locations during recovery. The function `abstractIn` retains only the mappings for the source locations ('loc') in the `fips` in its argument set. Thus, in PreFail, a tester has the power and flexibility to decide how to characterize and cluster recovery paths.

If we use the equivalence function in Figure 15.10 to cluster failure sequences that result in the same recovery path into the same class, then we would obtain four different equivalence classes for the HDFS write protocol. The third column in Figure 15.1 shows the four classes: □, ■, ●, and ○ which represent the recovery paths {ABCDE}, {CDEF}, {CDEG}, and {CDE} respectively. Note that the recovery paths of $A1_c$ and $B2_c$ are considered to be equivalent (□) as they have I/Os at the same set of source locations {ABCDE} even if the I/Os are executed in different nodes. However, if the tester decides to characterize recovery paths using both source location and node ID, then the recovery paths of $A1_c$ and $B2_c$ would be considered to be different (□ and △), as shown in the last column in Table 15.1.

Figure 15.11 provides more details of how different I/Os shown in Figure 15.5 are grouped into different recovery classes. The symbols (*e.g.*, A1, A2) represent I/Os described in Figure 15.5. A shape (*e.g.*, □) surrounding an I/O #X represents the equivalence class

```
1 def cls (fs1, fs2):
2  last1 = fs1 [ len(fs1) - 1 ]
3  last2 = fs2 [ len(fs2) - 1 ]
4  prefix1 = fs1 [ 0 : len(fs1) - 1 ]
5  prefix2 = fs2 [ 0 : len(fs2) - 1 ]
6  isEqv = eqvBySrcLoc (prefix1, prefix2)
7  return isEqv and (last1 == last2)
```

Figure 15.12: Equivalent-recovery clustering: Cluster two failure sequences if their prefixes result in the same recovery path and their last `fit`s are the same

of the I/O with regard to the recovery path that is taken by HDFS when a crash occurs at that I/O. Different shapes represent different equivalence classes. The left figure shows 4 recovery classes that result from the use of only source location to distinguish between different recovery paths. Even by just using source location, PreFail is able to distinguish between the two main recovery classes in the protocol (□ and ○). Furthermore, PreFail also finds two unique cases of failures that result in two more recovery classes (■ and ●). In the first one (■), a crash at C1 leaves the surviving nodes (`N2` and `N3`) with zero-length blocks, and thus the recovery protocol executes I/Os at a different source location (labeled with I in Table 15.1). In the second one (●), a crash at C2 leaves the surviving nodes (`N1` and `N3`) with different block sizes (the first node has received the bytes, but not the last node), and thus I/Os at yet another source location (labeled as J) are executed.

Figure 15.11b shows the 8 recovery classes that result when node ID is used in addition to the source location to characterize recovery paths. If the tester uses all of the context present in a `fip`, the I/Os in the write protocol will be grouped into 10 recovery classes. In general, the more context information in `fip`s considered, the more we can distinguish between different recovery paths, and hence the more the number of recovery classes of I/Os. Lesser context leads to fewer recovery classes and thus fewer failure-injection experiments, but might miss some corner-case bugs.

Clustering Failure Sequences

After specifying the characterization of a recovery path, the tester can simply write a cluster policy that uses the `cls` function in Figure 15.12. Given this policy, if there are two sequences, `(prefix1, last)` and `(prefix2, last)`, such that `prefix1` and `prefix2` result in the same recovery path, then PreFail will exercise only one of the two sequences. The `cls` function uses `eqvBySrcLoc` to compute the equivalence of the recovery paths of the prefixes.

To illustrate the result of this policy, let's consider the example in Table 15.1. The `fit` F_c (crash at I/O F) can be exercised after any of the crashes at $\{DE\}\times\{123\}$ (*i.e.*, 6 `fits`). Without the specified equivalent-recovery clustering, PreFail will run 6 experiments ($D1_cF_c$.. $E3_cF_c$). But with this policy, PreFail will group all of the 6 failure sequences into a single class ($D1_c$/../$E1_c$ + F_c) as all the prefixes have the same recovery class (○, as

```
1 def flt (fs):
2  for f in fs:
3    fp = FIP(f)
4    isCrash = (fp['failure'] == 'crash')
5    isWrite = (fp['ioType'] == 'write')
6    isBefore = (fp['place'] == 'before')
7    if isCrash and
            (not (isWrite and isBefore)):
8       return False
9  return True
```

Figure 15.13: Generic crash optimization: The function accepts a failure sequence if all crashes in the sequence are injected before write I/Os

shown in Figure 15.11), and thus will run only 1 experiment to exercise any of the 6 failure sequences. If the tester changes the clustering function such that it uses both source location and node ID to characterize a recovery path (Figure 15.11b), then PreFail will run three experiments as the prefixes now fall into three different recovery classes (○, ▽, and ▼).

15.5.5 Pruning via Optimizations

In general, failures can be injected before and/or after every read and write I/O, system call or library call. For some types of failures like crashes or disk failures, there are optimizations that can be performed to eliminate unnecessary failure-injection experiments. In the following sections, we present policies that implement optimizations for crashes and disk failures in distributed systems. By reducing the number of individual failure-injection tasks, these optimizations also help in reducing the number of multiple-failure sequences.

Crashes

In a distributed system, read I/Os performed by a node affect only the local state of the node, while write I/Os potentially affect the states and execution of other nodes. Therefore, we do not need to explore crashing of nodes around read I/Os. We can just explore crashing of nodes before write I/Os. Figure 15.13 shows a `flt` function that can be used to implement this optimization.

The second optimization that we can do for crashes is that we do not crash a node before the node performs a network write I/O that sends a message to an already crashed node. This is because crashing a node before a network write I/O can only affect the node to which the message is being sent, but the receiver node is itself dead in this case. The `flt` function that implements this optimization is shown in Figure 15.14. The function accepts a failure sequence if for each crash at a network write to a receiver node `rNode` in the sequence, there is no preceding crash in the sequence that occurs in the node `rNode`. The function uses the function `nodeAlreadyCrashed` (also implemented by the tester but not shown) that takes a

```
1 def flt (fs):
2  for i in range(len(fs)):
3    fp = FIP(fs[i])
4    isNet = (fp['ioTarget'] == 'net')
5    isWrite = (fp['ioType'] == 'write')
6    isCrash = (fp['failure'] == 'crash')
7    rNode = fp['receiver']
8    pfx = fs[0:i]
9    if isNet and isWrite and isCrash and
10      nodeAlreadyCrashed(pfx, rNode):
11      return False
12 return True
```

Figure 15.14: Crash optimization for network writes

failure sequence and a node as arguments, and returns true if there is a crash failure in the sequence that occurs in the given node.

Disk Failures

For disk failures (permanent and transient), we inject failures before every write I/O call, but *not* before *every* read I/O call. Consider two adjacent Java read I/Os from the same input file (*e.g.*, `f.readInt()` and `f.readLong()`). It is unlikely that the second call throws an I/O exception, but not the first one. This is because the file is typically already buffered by the OS. Thus, if there is a disk failure, it is more likely the case that an exception is already thrown by the first call. Thus, we can optimize and only inject read disk failures on the first read of every file (*i.e.*, we assume that files are always buffered after the first read). The subsequent reads to the file will naturally fail. The policy for this optimization is similar to the one for network failure optimization (Figure 15.15) as explained in the next section.

Network Failures

For network failures, we can perform an optimization similar to disk failures. Since there is no notion of file in network I/Os, we keep information about the latest network read that a thread of a node performs. If a particular thread performs a read call that has the same sender as the previous call, then we assume that it is a subsequent read on the same network message from the same sender to this thread (potentially buffered by the OS), and thus we do not explicitly inject a network failure on this subsequent read. In addition, we clear the read history if the node performs a network write, so that we can inject network failures when the node performs future reads on different network messages. Also, we do not inject a network failure if one of the nodes participating in the message is already dead. The `flt` function that can be used to implement the optimization for network failures is shown in

```
1 def flt (fs):
2  for i in range(len(fs)):
3    fp = FIP(fs[i])
4    isNetFail = (fp['failure'] == 'netfail')
5    isRead = (fp['ioType'] == 'read')
6    sender = fp['sender']
7    node = fp['node']
8    thread = fp['thread']
9    time = fp['time']
10   pfx = fs[0:i]
11   allFS = allFitSeqs()
12   if isNetFail and isRead and
13      (not first(pfx, node, thread, time, sender, allFS)):
14      return False
15 return True
```

Figure 15.15: Network failure optimization

Figure 15.15. The function checks for each network failure at a read I/O in a failure sequence to see if it is the first read of data in its thread that is sent by its sender to its node. It uses the function `first` (implemented by the tester, but not shown) to determine if a network read I/O is the first read of data in the read's thread that is sent by the data's sender to the receiver node. The key `time` in a `fip` records the time when the `fip` was observed during execution in the FI engine. This key helps in determining the temporal position of a read in the list of all failure sequences `allFS` passed on by the FI engine.

Disk Corruption

In the case of disk corruption, after data gets corrupted, all reads of the data give unexpected values for the data. It is possible but very unlikely that the first read of the data gives a non-corrupt value and the second read in the near future gives a corrupt one. Thus, we can perform an optimization similar to the disk-failure case.

15.5.6 Failing Probabilistically

Finally, a tester can inject multiple failures if they satisfy some probabilistic criteria. We have not explored this strategy in great extent because we need some real-world failure statistics to perform real evaluation. However, we believe that specifying this type of policy in PreFail will be straightforward. For example, the tester can write a policy as simple as: return true if `prob`(fs) > 0.1. That is, inject a failure sequence *fs* only if the probability of the failures happening together is larger than 0.1. The tester needs to implement the `prob` function that ideally uses some real-world failure statistic (*e.g.*, a statistic that shows the probability distribution of two machine crashes happening at the same time).

In summary, the programmable policy framework allows testers to write various failure exploration policies in order to achieve different testing and optimization objectives. In addition, as different systems and workloads employ different recovery strategies, we believe this programmability is valuable in terms of systematically exploring failures that are appropriate for each strategy. In the next two chapters, we explain how we have implemented our programmable testing tools (PreFail and PCheck), and also evaluate the effectiveness and usefulness of policies used in the tools for testing real-world distributed systems.

Chapter 16

Implementation

We explain how we have implemented our programmable testing tools. Both PCheck (Chapter 14) and PreFail (Chapter 15) instrument the distributed system under consideration and add hooks to it so that they can observe the execution of relevant operations or events (*e.g.*, message receive events and system calls). Using the instrumentation hook that is inserted into the system to observe a relevant event, the testing tool can implement what it needs to do when it observes that event. For example, PCheck would block the execution of the relevant event, and would let it proceed at a later time to enforce a particular message ordering that it wants to explore, and PreFail would execute code to emulate a failure (*e.g.*, throwing of an exception and network failure) during (or before or after) the execution of the relevant event. The hook gathers appropriate context information (*e.g.*, source location, call stack, network message content etc.) related to the relevant event, and passes it on to the tool so that the tool can use the information to implement its action. We first describe how we instrument the system under test, and then describe the policy frameworks of PCheck and PreFail in more detail. Our tools have been built for systems written in Java.

16.1 The instrumentor

We instrument the system under test to add hooks to observe relevant events and gather context information regarding those events. The MC Facilitator in PCheck and the FI engine in PreFail are responsible for instrumenting the system and observing relevant events. In both the tools, we use AspectJ [3] to instrument the system. We weave the Java bytecode of the system with aspects that track the execution of relevant events and collect context for those events. AspectJ allows us to specify execution points of interest like execution of particular method calls or instantiation of particular objects. We can add an aspect for an execution point of interest so that when that point is reached during execution, the aspect can gather the context information at that point and send it to the testing tool. For example, the aspect below adds a hook to observe the read from an input stream.

```
Object around(): (call (* InputStream+.read*(..))){ ... }
```

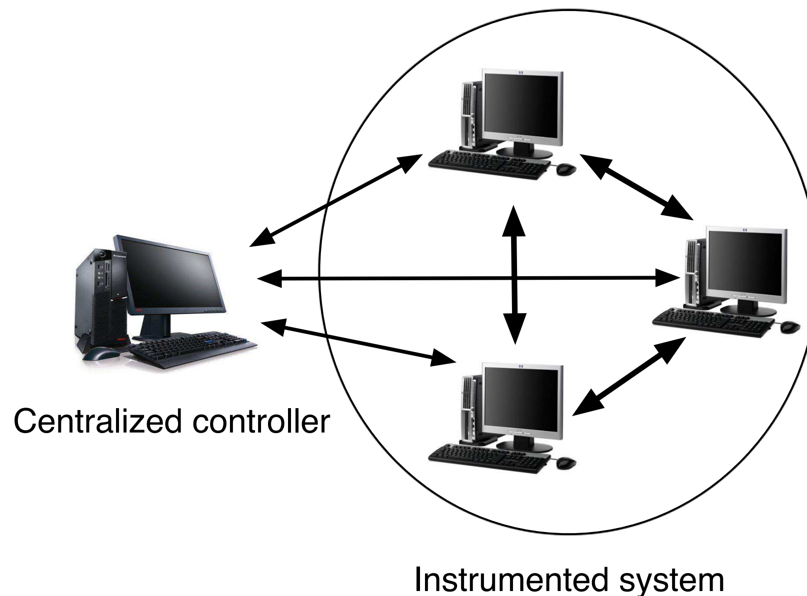


Figure 16.1: Architecture of an instrumented system

This aspect would be executed whenever an input stream is going to be read. The aspect can send the relevant context information associated with the read (*e.g.*, the node where the read occurs, the call stack when the read occurs etc.) to the testing tool that can then perform a suitable action for the read call. For example, the tool can block the read (as in PCheck), or it can inject an exception during read (as in PreFail).

Figure 16.1 shows how an instrumented system works. Whenever an inserted hook is executed in a node in the system, the hook gathers the relevant context information and passes it on using RPC to a centralized controller (the MC Facilitator in Chapter 14 or the FI engine in Chapter 15) which is another node or process. The controller, which is part of the testing tool, can then decide to perform an appropriate action in response to the event whose context information was sent by the instrumented node. The node waits for the controller to finish its action before proceeding with its execution. The advantage of having a centralized controller is that the controller has a global view of the events in the entire system, and thus can take better decisions about its actions. For example, if the tester wants to reorder messages across different nodes, the centralized controller can block message receive events in different nodes and unblock them in the order in which the tester wants them to execute. If we do not have a centralized controller, then we cannot know about the messages being received at other nodes, and hence, we cannot force an ordering over messages at different nodes. Similarly, if the tester wants to inject two failures during execution, then without a centralized controller, the testing tool would not be able to track the number of failures that have already been injected across all nodes.

16.1.1 Instrumentation in PCheck

In PCheck, since we are interested in testing the correctness of a system under different message orders, we instrument message receive statements so that we can control the order in which the network messages are received and processed. We instrument those statements that can receive data from the network (*e.g.*, reading from socket input streams) so that the MC Facilitator can block the execution of those statements and unblock them in a manner that imposes an ordering on them. To block a message receive event, the Facilitator executes a wait on a monitor. It waits on that monitor until the MC Engine directs it to unblock that event. To unblock the event, the Facilitator notifies the monitor on which it was waiting.

In addition to instrumenting statements that receive network messages, we also need to instrument the statements that assemble the packets in those messages so that we can expose the contents of the packets to the testers who can use them in their policies. Instrumenting the statements that assemble message packets requires identifying those statements which can differ from one system to another. We currently require the tester to manually identify the statements and extend the instrumentation framework to track those statements. Even if it might look like a big challenge to identify and instrument packet assembling statements, in our experience, these statements are concentrated across a few functions and not scattered around. We could easily port PCheck to three different systems (ZooKeeper [61], Cassandra [75], and Hadoop File System [100]) by instrumenting only 25, 14, and 24 statements respectively to expose packet contents to policy writers.

We give an example of the statements that we have to instrument to expose the contents of the network messages to testers. In the leader election protocol in ZooKeeper, we have to instrument the following statement to obtain the different fields of the network message that was received at a socket. The message is being put into the receive queue (`recvQueue`) here so that it can be processed later.

```
recvQueue.put(new Message(message.duplicate(), sid));
```

The reason we have to instrument this statement in addition to the low-level socket communication is that we do not know how to interpret the data read from a socket. The data read is usually just a stream of bytes, and unless it gets reconstructed into a meaningful object, it is hard to understand the data. In the above function call, the data read off the socket has been reconstructed into a `Message` object, and is being passed as an argument. We can access the argument, and read its fields to understand the network data that we had earlier read from the socket.

16.1.2 Instrumentation in PreFail

PreFail instruments system or library calls for performing disk or network I/Os [50] (*e.g.*, file reads and writes, reads from socket input streams, and writes to socket output streams) since a failure in the hardware would manifest as a failure during an I/O. For example, if the

network link between two nodes fails, all writes to and reads from that link would fail and throw exceptions. Thus, we can inject (that is, emulate) a network link failure by throwing exceptions at system or library calls that write from and read to that link. We can similarly inject other kinds of failures by failing the relevant I/Os in the right manner. For example, we can inject a disk failure by throwing exceptions at reads from and writes to the disk, or we can inject a node failure by terminating the process running on that node. Some of these failures can be transient or persistent. For example, a network link might be down for sometime before coming back up, or a disk might have a temporary failure. We can inject both transient and persistent failures. To inject a transient failure, we throw an exception once for a system or library call, and to inject a persistent failure, we throw an exception whenever the call is executed.

Unlike PCheck, we do not require very system specific information to implement the instrumentation in PreFail. This is because the kinds of failures that we can inject at a system or library call are independent of how the call occurs in a system. Thus, the instrumentation to inject the failures at the system or the library call would not vary from system to system.

16.2 The policy framework

Programmable testing tools provide a framework to testers in which the testers can easily express their insights and knowledge to guide the testing process. For example, the MC Driver in PCheck provides a framework with network message related abstractions that the testers can use to express which messages they want to reorder and which they do not want to reorder. Similarly, the FI driver provides support for expressing the failures and the failure combinations that the testers want to test. To design a policy framework, one has to decide which abstractions to provide to the testers so that they can easily and quickly express their intuitions and intentions in the form of testing policies. We describe the abstractions that we provide in our programmable tools, MC and PreFail, below. The policy frameworks for both the tools are implemented in Python.

16.2.1 Abstractions in PCheck

We provide abstractions for network messages in PCheck that can be used by testers in their policies. An abstraction for a network message is a Python object whose fields represent the context and contents of the message. For example, the field named ‘sender’ is the ID of the node that sent the message, ‘receiver’ is the ID of the node that received the message, ‘stack’ is the call stack at the sender node when the message was sent, ‘vote’ is the ID of the node for which the sender node is voting in the leader election protocol, and ‘type’ is the type of the message (‘request’, ‘ack’, ‘proposal’, etc.). Testers can access the different fields of messages, and compare them against each other in their policies. For instance, a tester might compare the ‘vote’ fields of two messages to determine if the received messages are votes for different

nodes or not. The abstractions provided by the MC Driver to the testers are the same as the abstractions provided by the MC Facilitator to the MC Engine, but are implemented in a high-level scripting language (Python in our case) to make it easier for testers to write their policies. Testers implement a policy by implementing a specific function (the `dep` function) that takes two messages and a sequence of messages as arguments and returns a boolean. The implementation of the function should decide if the two given messages should be considered to be dependent (and hence should be reordered) in the system state that results from the processing of the given sequence of messages.

16.2.2 Abstractions in PreFail

In PreFail, we provide abstractions for failures that can be injected during execution. An abstraction for a failure is again a Python object whose fields represent the type of the failure and the context associated with the execution point where the failure can be injected. For example, the field 'type' represents the type of the failure that can be injected, 'node' represents the ID of the node where the failure can be injected, 'stack' represents the call stack at the node where the failure can be injected, and 'disk' represents the ID of the disk that might fail if the failure is injected. We also provide abstractions for execution points where failures can be injected, that is, failure-injection points (Chapter 13). These abstractions are similar to the failure abstractions. They just leave out the fields that specify failure-related information (*e.g.*, the 'type' field).

Testers can access the different fields of failure abstractions to decide if they want to exercise the failures or not. To implement a policy, the tester can implement the `flt` function or the `cls` function that are two specific functions recognized by the policy framework. The `flt` function takes a sequence of failure abstractions as argument and returns a boolean. The implementation of the `flt` function by the tester should decide whether to exercise the failure sequence represented by the argument. The tester can use the different fields of the failure abstractions in the given sequence to decide whether to exercise the failure sequence or not. For example, if the tester wants to inject only crashes, then she can implement the `flt` function to return true only when the 'type' of all the failures in the sequence is 'crash'. The `cls` function takes two sequences of failure abstractions as arguments and returns a boolean. The implementation of the `cls` function should decide if the given failure sequences should be considered to be equivalent or not. Again, the tester can use the fields of the failure abstractions in her implementation of the `cls` function.

The policy framework also provides a library of functions that help the tester to access the profiles of executions that have already been executed. For example, the `allFips` function takes a sequence of failure abstractions as argument and returns the set of abstractions of failure-injection points that were observed in the execution in which the argument failure sequence was injected. Similarly, we also have the `postInjectionFips` function that given a failure sequence returns the set of failure-injection points that were observed after the given sequence was injected in a previous system execution.

16.3 Tool complexity

In PCheck, the MC Engine is written in Java, and has 9.5K LOC. The instrumentation code in MC Facilitator varies from system to system; it is 1892, 1314, and 649, respectively for ZooKeeper, Cassandra, and HDFS. The FI engine in PreFail is based on the failure-injection tool in [50], which is written in 6000 lines of Java code. We added around 160 lines of code to this old tool so that it passes on appropriate failure and execution abstractions to the FI driver. The FI driver is implemented in 1266 lines of Python code. The source code for PCheck can be found at `https://github.com/pallavij/PCheck`, and the source code for PreFail is released at `http://cloudfail.cs.berkeley.edu/download.html`.

In the next chapter, we evaluate our tools, PCheck and PreFail, on popular real-world distributed systems.

Chapter 17

Evaluation

In this chapter, we provide detailed evaluation of various aspects of our programmable tools, PCheck and PreFail. We have implemented the tools to test distributed systems written in Java. We experimented with three different systems: ZooKeeper [61], a system that provides distributed synchronization services like leader election among other services that can be used by other distributed systems; Cassandra [75], a distributed database, and Hadoop File System [100] (HDFS), a distributed file system. For all of the systems, we wrote different workloads to exercise different protocols within the systems. We explain the workloads in subsequent sections. We next discuss how the policies that we wrote for different systems and workloads reduced the state space (of message orderings or failure sequences) that we had to explore yet helped us to achieve our testing objectives.

17.1 Effectiveness of Policies

In this section, we show how policies are effective in achieving testing objectives while reducing the state space to explore.

17.1.1 Policies in PCheck

For testing event (network message in our work) orderings, our testing objective was branch coverage. Thus, we want to compare the branch coverage that we obtain using policies with the coverage that we obtain with state-of-the-art testing approaches. The workloads that we wrote for PCheck include workloads that exercise two different leader election protocols within ZooKeeper, the client write protocol after a leader has been elected in ZooKeeper, the gossiper protocol in Cassandra that helps nodes in the system to keep track of other alive nodes, the insert protocol in Cassandra to insert data into the database, and the file write and append protocols in the Hadoop File System. For Hadoop File System, we used the version provided by Cloudera.

Workload	# m	Policy	# execs	Speed-up	Branch cov.
ZK-LE1	60	Policy 1	65	6.7	77.97%
		Policy 2	100	4.36	76.27%
		Random	500	-	77.97%
		Normal	500	-	69.49%
ZK-LE2	29	Policy 1	27	3.56	44.9%
		Policy 2	52	1.85	44.9%
		Random	500	-	44.9%
		Normal	500	-	43.9%
ZK-CW	12	Policy 1	4	5.75	43.7%
		Random	500	-	43.7%
		Normal	500	-	43.7%
CASS-GOSS	21	Policy 1	6	28.6	45.4%
		Random	500	-	45.4%
		Normal	500	-	45.4%
CASS-CI	4	Policy1	1	2	25.7%
		Random	500	-	25.7%
		Normal	500	-	26.5%
HDFS-WR	27	Policy 1	2	2	25.93%
		Random	500	-	25.93%
		Normal	500	-	18.5%
HDFS-APP	27	Policy 1	2	2.5	27.1%
		Random	500	-	27.1%
		Normal	500	-	19.75%

Table 17.1: Branch coverage with policies. Here, N = 1, that is, we allow reordering of at most one pair of events in each execution

Table 17.1 shows how branch coverage varies with the system executions that we explore with different policies. The first column is the name of the system and the workload used. The first three workloads are for ZooKeeper (ZK): LE1 and LE2 are two versions of the leader election protocol and CW is the client write protocol. The next two workloads are for Cassandra (CASS): the first one is the gossiper protocol and the second one is the database insert protocol. The last two workloads are the write and the append protocols for HDFS. All the workloads were run on systems consisting of three nodes. The second column is the average number of messages observed when a workload is executed. Note that not all messages are "concurrent", that is, a message cannot be ordered with all of the other messages.

The third column in Table 17.1 provides the various policies that we wrote for the workloads. We have presented some of the policies in Chapter 14. The `Random` policy chooses random event orderings without paying any attention to the orderings that were explored in previous executions. At each stable state during execution, the policy randomly chooses one of the blocked events and releases it. Thus, the `Random` policy corresponds to random

testing which is the state-of-the-art when it comes to testing complex large-scale systems. The `Normal` policy executes the system as it would without interfering with how events order during execution, that is, the policy lets the execution resolve non-determinism in event ordering on its own. Testers often test their systems by simply executing them repeatedly and hoping that different executions would result in different event orderings. We wrote specific policies (Policy *) for each workload based on our understanding of the workload and the system. The fourth column is the number of system executions that we explore with each policy, and the fifth column is the speed-up in testing that we achieve with a policy. Speed-up here is defined as the number of executions we would explore with model checking with partial-order reduction divided by the number of executions we explore with a policy. The last column provides the branch coverage achieved with a set of executions.

Table 17.1 shows that in most of the workloads, we quickly reached the branch coverage obtained by the `Normal` policy and sometimes even did better (e.g., ZK-LE1 and HDFS-WR) in just a few executions with the specific policies written for the workloads. We also did as well as random testing, and in fact, we obtained the same coverage with fewer executions using policies than using random testing as shown later (Figures 17.1 and 17.2). The coverage with random testing can vary from one set of executions to the other, but we can more reliably obtain the same level of coverage with a policy each time. For CASS-CI, HDFS-WR, and HDFS-APP, the messaging that we observed was mostly synchronous, that is, messages would be sent only after previous messages have been processed, and thus, there were few concurrent messages during execution. As a result, the number of orderings of relevant concurrent messages that we could perform was low for each of those workloads. If we have a better knowledge of the systems and workloads, then we can instrument and add more relevant context to the messages and can prune down the execution space more precisely to obtain higher speed-ups.

Figures 17.1 and 17.2 illustrate how branch coverage increases with the number of executions explored. Figure 17.1 shows how the two specific policies for the leader election workload ZK-LE1 quickly reach (and even surpass) the coverage attained by random and normal testing. This shows that we can achieve our testing objectives with fewer executions with appropriate policies. Figure 17.2 shows the same trend for policies for the other leader election protocol ZK-LE2.

Table 17.2 illustrates how the number of explored executions and branch coverage are affected by increasing N which is the tester-provided parameter to restrict the number of reorderings. When we increase N to 2, the number of executions increases, but the speed-up is also much higher (25X or more for about half of the policies). The branch coverage is the same or more as compared to $N = 1$ for all the policies.

17.1.2 Policies in PreFail

For failure testing, we wrote four different workloads for the Hadoop File System to test its file read, write, and append protocols, and its recovery protocol using edit logs, two workloads for Cassandra to test its database insert protocol and recovery protocol using logs,

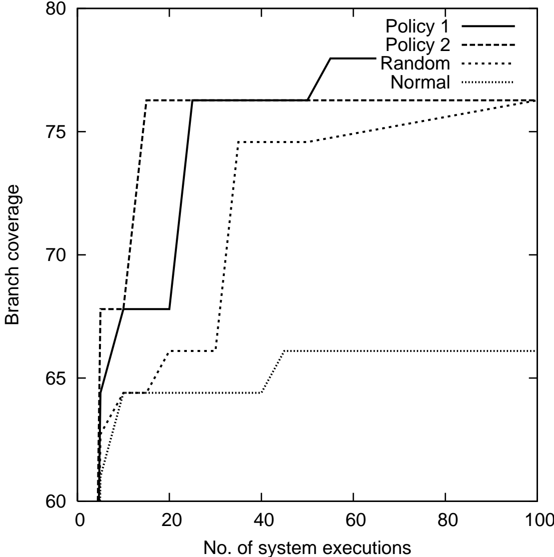


Figure 17.1: Policies for ZK-LE1

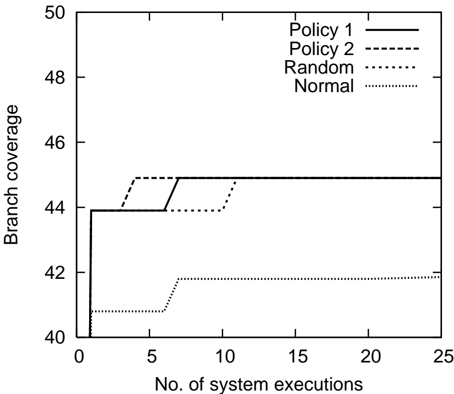


Figure 17.2: Policies for ZK-LE2

Workload	Policy	# executions	Speed-up	Branch cov.
ZK-LE1	Policy 1	121	177.3	77.97%
ZK-LE1	Policy 2	746	28.76	77.97%
ZK-LE2	Policy 1	32	25	44.9%
ZK-LE2	Policy 2	63	12.7	44.9%
ZK-CW	Policy 1	6	17.5	43.7%
CASS-GOSS	Policy 1	6	61	45.4%
CASS-CI	Policy 1	1	2	25.7%
HDFS-WR	Policy 1	2	3	25.93%
HDFS-APP	Policy 1	2	3	27.1%

Table 17.2: Effect of increasing N. Here, N = 2, that is, we allow reordering of at most two pairs of events in each execution

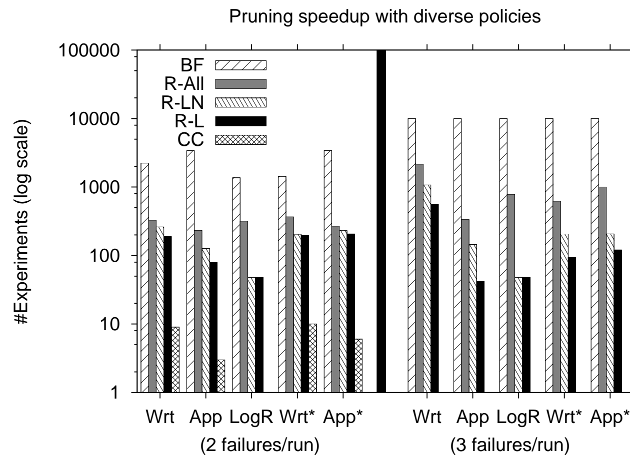


Figure 17.3: #Experiments run with different coverage-based policies

and one protocol for ZooKeeper to test one of its leader election protocols. We first present the evaluation of the code coverage (Section 15.5.3) and recovery coverage (Section 15.5.4) policies, and then the optimization-based policies (Section 15.5.5).

Coverage-Based Policies

We show the benefits of using different coverage-based failure exploration policies to prune down the failure space in different ways. Figure 17.3 shows the different number of experiments that PreFail runs for different policies. An experiment takes between 5 to 9 seconds to run. Here, we inject only crashes so that the numbers are easy to compare. The figure only shows numbers for multiple-failure experiments because injecting multiple failures is where the major bottleneck is. The y-axis shows the number of failure-injection experiments for a

Workload	**#F**	**#Failed Exps**	**#Bugs**	**#Bugs$_R$**
Write	2	0	0	0
	3	46	1	1
Append	2	14	2	2
	3	31	(*) 2	(*) 2
LogRecovery	2	6	3	0
	3	3	(*) 3	0

Table 17.3: #Bugs found. (*) implies that these are the same bugs (*i.e.*, bugs in 2-failure cases often appear again in 3-failure cases).

given policy and workload. The x-axis shows the workloads: the write (Wrt), append (App), and log recovery (LogR) protocols from Cloudera's version of HDFS. We also run workloads from an old HDFS release v0.20.0 (marked with *), which has a different design (and hence different results). Two and three crashes were injected per experiment for the bars on the left- and right-hand sides respectively. CC and BF represent the code-coverage policy and brute-force exploration, respectively. R-L, R-LN, and R-All represent recovery-coverage policies that use three different ways to characterize recovery (§15.5.3): using source location only (*L*), source location and node (*LN*), and all information in `fip` (*All*). We stopped our experiments when they reached 10,000 (Hence, the maximum number of experiments is 10,000).

With PreFail, a tester can choose different policies, and hence different numbers of experiments and speed-ups, depending on her time and resource constraints. For example, the code-coverage policy (CC) gives two orders of magnitude improvement over the brute-force approach because it simply explores possible crashes at source locations that it has not exercised before (*e.g.*, after exploring two crashes, there is no new source location to cover in 3-crash cases). Recovery clustering policies (R-L, R-LN, etc.) on the other hand run more experiments, but still give an order of magnitude improvement over the brute-force approach. The more relaxed the recovery characterization, the lesser the number of experiments (*e.g.*, R-L vs. R-All).

Pruning is not beneficial if it is not effective in finding bugs. In our experience, the recovery clustering policies are effective enough in rapidly finding important bugs in the system. To capture recovery bugs in the system, we wrote simple recovery specifications for every target workload. For example, for HDFS write, we can write a specification that says "if a crash happens during the data transfer stage, there should be two surviving replicas at the end". If a specification is not met, the corresponding experiment is marked as failed.

Table 17.3 shows the number of bugs that we found even with the use of the most relaxed recovery clustering policy (R-L, which only uses source location to characterize recovery). The table shows the number of crashes per run (#F) along with the actual number of bugs that trigger the failed experiments (#Bugs). The last column (#Bugs$_R$) is the number of

Workload	Crash	Disk Failure	Net Failure	Data Corruption
H. Read	2/42	1/4	4/17	1/4
H. Write	57/454	27/27*	45/200	N.A.
H. Append	111/880	43/60	117/380	1/18
H. LogR	36/128	39/64	N.A.	3/28
C. Insert	33/102	25/25*	12/26	N.A.
C. LogR	84/196	89/98	N.A.	5/14
Z. Leader	39/132	21/21*	31/45	N.A.

Table 17.4: Benefits of Optimization-based Policies. (*) These write workloads do not perform any disk read, and thus the optimization does not work here.

bugs that can be found using randomized failure injection, that is, by randomly choosing the execution points at which to inject crashes. For each workload, we execute the system as many times as we do for the recovery clustering policy, and randomly inject crashes in each execution. Randomized failure injection can find the bugs for the write and append workloads, but not for the log recovery workload. This is because the bugs for the log recovery workload are corner-case bugs; the proportion of failure sequences that lead to a log recovery bug is much smaller than that for a write or an append bug. This shows that randomized failure injection (state-of-the-art in testing multiple failures), though simple to implement, is not effective in finding corner-case bugs that manifest only in specific failure scenarios.

Optimization-Based Policies

Table 17.4 shows the effectiveness of the optimizations of different failure types that we described in Section 15.5.5. H in the table is for HDFS, C is for Cassandra, and Z is for ZooKeeper. Each cell presents two numbers X/Y where Y and X are the numbers of failure-injection experiments for single failures without using and with using the optimization respectively. N.A. represents a not applicable case; the failure type never occurs for the workload. For write workloads, the replication factor is 3 (*i.e.*, 3 participating nodes). Overall, depending on the workload, the optimizations bring 21 to 1 times (5 on average) of reduction in the number of failure-injection experiments.

17.2 Ease of writing policies

We wrote policies for all of our benchmark systems and workloads: two each for the leader election workloads and one each for the rest of the workloads in PCheck, and three recovery-coverage policies, one code-coverage policy, and four optimization-based policies for each of the workloads in PreFail. In PCheck, writing a policy and instrumenting to expose

relevant context for network messages required some knowledge about the system and the workload, but we could gain this knowledge from browsing relevant documentation and source code. Someone with a better knowledge of the system and the workload can write more specific and precise policies and can also instrument source code to expose more context for network messages. In PreFail, we did not require much system and workload specific knowledge for the policies that we wrote. The coverage-based policies are generic, and can be applied to other systems. The optimization-based policies also do not require any system specific knowledge. On an average, the size of a policy was 19 lines of code in PCheck, and 17 lines of code in PreFail.

17.3 Bugs found

We explain the recovery bugs that we found using PreFail. We were able to find all of the 16 bugs in HDFS v0.20.0 (older version of HDFS) that we had reported in our previous work [50]. We were told that many internal designs of HDFS have changed since that version. After we integrated PreFail to a much newer HDFS version (the Cloudera version v0.20.2+737), we found 6 more previously unknown bugs (three have been confirmed, and three are still under consideration). Importantly, the developers believe that the bugs are crucial ones and are hard to find without a multiple-failure testing tool. These bugs are basically availability (*e.g.*, the HDFS master node is unable to reboot permanently) and reliability bugs (*e.g.*, user data is permanently lost). We explain below one of the new recovery bugs. This bug is present in the HDFS append protocol, and it happens because of multiple failures.

The task of the append protocol is to atomically append new bytes to three replicas of a file that are stored in three nodes. With two node failures and three replicas, append should be successful as there is still one working replica. However, we found a recovery bug when two failures were injected; the append protocol returns error to the caller and the surviving replica (that has the old bytes) is inaccessible. Here are the events that lead to the bug: the first crash causes the append protocol to initiate a quite complex distributed recovery protocol. Somewhere in the middle of this recovery, a second crash happens, which leaves the system in an unclean state. The protocol then initiates another recovery. However, since the previous recovery did not finish and the system state was not properly cleaned, this last initiation of recovery (which should be successful) cannot proceed. Thus, an error is returned to the append caller, and since the surviving replica is in an unclean state, the file cannot even be accessed.

In summary, we have seen in our experiments that policies can reduce the state space to explore significantly yet achieve testing objectives like branch coverage or bug coverage. Using our programmable tools, testers can write a variety of policies without having any knowledge about the internals of the underlying testing process.

Chapter 18

Other related work

We have mentioned and compared our programmable tools with related work at various points in the previous chapters. In this chapter, we describe more related work and compare our tools with them.

Model checking of distributed systems: MoDist [114] interposes a thin layer between the system being considered and a model checker. The layer intercepts various OS operations during execution and passes them on to the model checker that then performs a full-fledged model checking on sequences of those operations. MoDist incorporates partial-order reduction, but the reduction is traditional and is based on using synchronization of two operations with respect to each other to determine if they are dependent. We use tester-written policies to determine if two operations are dependent in PCheck. Even if two operations are deemed dependent according to their synchronization with respect to each other, the tester might not want to test both orders of the operations as she might know that testing either order would suffice for her testing objectives. Thus, using policies we can obtain a further reduction in the number of executions that we have to explore as compared to model checking with traditional partial-order reduction. Moreover, MoDist intercepts and considers all kinds of OS operations, but PCheck restricts itself to only those operations that are interesting or relevant to the tester. As a result PCheck is faster, but it has to tolerate non-determinism of non-relevant operations.

DeMeter [51] uses interface reduction to reduce the state-space of model checking. It dynamically finds out the interface according to which a component in a system interacts with the other components, and using this information it locally model checks each component. DeMeter and PCheck share the same goal of reducing state space, but DeMeter achieves this by automatically dividing the execution into global and local component interactions and exhaustively checking the local interactions and PCheck achieves this by putting more power in the hands of the testers and by allowing them to use their knowledge and intuition to identify how they want to reduce the state space.

MaceMC [73] is a model checker for applications written in the Mace language. It tests all possible orders of the events identified in the application. FiSC [115] is another model checker that checks file systems but does not use partial-order reduction. Instead, it uses techniques

like using hashing to merge "similar" states. Such state-space reduction techniques are hard-coded into FiSC, but can be implemented by the tester in our approach according to when the tester considers two states or messages to be equivalent.

We did not find any model checker that reduces the space of multiple-failure combinations when performing failure testing as we do in PreFail by writing suitable policies.

Using high-level languages in testing: To the best of our knowledge, there is no prior work that provides a high-level language or framework to enable testers to influence model checking of distributed systems. But, there has been some previous work in designing high-level languages for failure testing of distributed systems or applications. LFI [77] provides a framework to allow testers to write down failure scenarios that occur during library calls. FIG [17] is another framework that lets testers control failure injection at the library level. Testers can specify which library calls do they want to fail, and with what frequency. Orchestra [28] uses Tcl scripts written by testers to fail or corrupt network messages. The Tcl scripts use TCP headers of messages to determine whether to fail them or not. In PCheck, rather than providing the low-level context (*e.g.*, TCP header) regarding a message to testers, we instrument and track more high-level or "semantic" information being carried in the message (*e.g.*, vote in the message) and expose them so that testers can use them in their policies. Genesis2 [70] allows testers to use scripts to control fault injection in Web services and clients, and FAIL-FCI [56] provides a domain-specific language in which testers can control failure injection in the Grid middleware. In these failure injection tools, even though a high-level or a domain-specific language is provided, testers often might have to write significant amount of code from scratch to pinpoint the execution context of the failure scenarios that they want to exercise. The event abstractions in PCheck and failure abstractions in PreFail on the other hand already capture the relevant context for events and failures. The context can thus be directly accessed and used by testers without writing a lot of code.

Pruning down state space: There has been some work in devising smart techniques that systematically prune down large search spaces. Extensible LFI [78] for example automatically analyzes the system to find code that is potentially buggy in its handling of failures (*e.g.*, system calls that do not check some error-codes that could be returned). AFEX [71] automatically figures out the set of failure scenarios that when explored can meet a certain given coverage criterion like a given level of code coverage. It uses a variation of stochastic beam search to find the failure scenarios that would have the maximal effect on the coverage criterion. Fu et. al. [44] use compile-time analysis to find which failure-injection points would lead to the execution of which error recovery code. They use this information to guide failure injection to obtain a high coverage of recovery code. To the best of our knowledge, the authors of these works do not address pruning of combinations of multiple failures in distributed systems.

There has been some work in program testing [15, 26, 47] that uses tester-written specifications or input generators to produce all non-isomorphic test inputs bounded by a given size. The specifications or generators can be thought of as being analogous to the tester-written policies, and the process of generating inputs from them by pruning down the input

space can be thought of as being analogous to the process of pruning down the space of all event orderings or the failure space using policies. The specifications are used only for the purpose of generating test inputs, and there is no support to address event orderings or failures in the specifications.

Randomized testing: This is the state-of-the-art when it comes to testing large complex distributed systems. For example, randomized injection of multiple failures is employed to test systems in Google [21], Yahoo! [105], Microsoft [114], Amazon [55], and Netflix [58]. Randomized testing is simple to implement, but it can miss corner-case bugs that can occur only when events (*e.g.*, network messages) order in a specific manner, or when specific failures occur at specific points of execution.

Chapter 19

Conclusion

Poor software reliability is a serious concern today. Software bugs have caused significant financial losses, accidents, and even deaths. The techniques explored in this thesis can help mitigate the problem by improving the effectiveness of software testing, specifically testing of concurrent and distributed systems that have become integral components of today's software systems. Testing is still the most widely used technique in the industry to validate software systems. Thus, improving the state-of-the-art of testing will improve the reliability of software systems.

In this thesis, we have built techniques that amplify the effectiveness of testing by not only exploring the program executions that showed up during testing but also by exploring executions that came "close" to happening. The first part of the thesis describes how the power of program analysis can be used in the background to build testing tools that provide the familiar interface that traditional testing tools provide but that can not only find bugs in the program executions that they observe but also *predict* and *confirm* bugs in "nearby" program executions. We have built predictive testing tools to find deadlocks (both resource and communication) and typestate errors in concurrent software systems. Our tools have found a number of serious bugs in real-world software systems. Even though our tools have been designed for deadlocks and typestate errors, some of the key ideas in them can be extended to find other kinds of bugs. For example, the trace program based approach (Chapter 5) can be extended to find other kinds of bugs by retaining the events necessary to find those kinds of bugs in the trace programs. DeadlockFuzzer (Chapter 7) can also be extended to confirm other classes of bugs (Section 7.5) by actively controlling the thread scheduler and pausing threads at suitable points to reproduce the real bugs.

We have also developed tools that allow testers to easily express their insights and knowledge regarding the programs under consideration to guide the underlying testing process towards interesting program executions that are more likely to help achieve the tester's objectives or are more likely to exhibit bugs. As described in the second part of the thesis, we have implemented *programmable* tools in which testers can express their intuitions and intents in a high-level language to improve the efficiency of testing of correctness and robustness against failures of large-scale distributed systems. We have some default policies

(expressions of intuition) in the programmable tools that can be used even when the testers do not have a detailed knowledge of the systems under consideration. The testers can however write better policies if they have a good understanding of the systems. They can even start with a specific policy that exercises only some specific program executions, and improve on the policy with the knowledge that they gain in the process of using it. We have found a number of recovery bugs in popular real-world distributed systems with the policies that we have written in our programmable tools.

Bibliography

[1] http://code.google.com/p/jchord/.

[2] Amazon EC2. http://aws.amazon.com/ec2.

[3] AspectJ. www.eclipse.org/aspectj.

[4] Hadoop MapReduce. http://hadoop.apache.org/mapreduce.

[5] R. Agarwal and S. D. Stoller. Run-time detection of potential deadlocks for programs with locks, semaphores, and condition variables. In *PADTAD*, pages 51–60, 2006.

[6] R. Agarwal, L. Wang, and S. D. Stoller. Detecting potential deadlocks with static analysis and runtime monitoring. In *PADTAD*, 2005.

[7] G. Ammons, R. Bodik, and J. R. Larus. Mining specifications. In *Symposium on Principles of Programming Languages*, 2002.

[8] C. Artho and A. Biere. Applying static analysis to large-scale, multi-threaded Java programs. In *Proceedings of the 13th Australian Software Engineering Conference (ASWEC'01)*, pages 68–75, 2001.

[9] P. Avgustinov, J. Tibble, and O. de Moor. Making trace monitors feasible. *SIGPLAN Not.*, 42(10):589–608, 2007.

[10] O. Babaoğlu and K. Marzullo. Consistent global states of distributed systems: Fundamental concepts and mechanisms. In S. Mullender, editor, *Distributed Systems*, pages 55–96. Addison-Wesley, 1993.

[11] S. Bensalem, J.-C. Fernandez, K. Havelund, and L. Mounier. Confirmation of deadlock potentials detected by runtime analysis. In *PADTAD*, 2006.

[12] S. Bensalem and K. Havelund. Dynamic deadlock analysis of multi-threaded programs. In *Haifa Verification Conference*, pages 208–223, 2005.

[13] S. Bensalem and K. Havelund. Scalable dynamic deadlock analysis of multi-threaded programs. In *PADTAD*, 2005.

[14] A. W. Biermann and J. A. Feldman. On the synthesis of finite-state machines from samples of their behavior. *IEEE Transactions on Computers*, pages 591–597, 1972.

[15] C. Boyapati, S. Khurshid, and D. Marinov. Korat: Automated Testing Based on Java Predicates. In *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA '02)*, pages 123–133, Rome, Italy, July 2002.

[16] C. Boyapati, R. Lee, and M. Rinard. Ownership types for safe programming: preventing data races and deadlocks. In *ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages and Applications*, pages 211–230, 2002.

[17] P. Broadwell, N. Sastry, and J. Traupman. FIG: A Prototype Tool for Online Verification of Recovery Mechanisms. In *Workshop on Self-Healing, Adaptive and Self-Managed Systems*.

[18] M. Burrows. The Chubby lock service for loosely-coupled distributed systems. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation (OSDI '06)*, Seattle, Washington, Nov. 2006.

[19] G. Candea and A. Fox. Crash-Only Software. In *The Ninth Workshop on Hot Topics in Operating Systems (HotOS IX)*, Lihue, Hawaii, May 2003.

[20] S. Chaki, E. Clarke, J. Ouaknine, N. Sharygina, and N. Sinha. Concurrent software verification with states, events, and deadlocks. *Formal Aspects of Computing*, 17(4):461–483, 2005.

[21] T. Chandra, R. Griesemer, and J. Redstone. Paxos Made Live - An Engineering Perspective. In *Proceedings of the 26th ACM Symposium on Principles of Distributed Computing (PODC '07)*, Portland, Oregon, Aug. 2007.

[22] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. Gruber. Bigtable: A Distributed Storage System for Structured Data. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation (OSDI '06)*, pages 205–218, Seattle, Washington, Nov. 2006.

[23] F. Chen, T. F. Serbănută, and G. Rosu. jpredictor: A predictive runtime analysis tool for java. In *International Conference on Software Engineering (ICSE'08)*. ACM press, 2008.

[24] 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 *Proc. of the ACM SIGPLAN Conference on Programming language design and implementation*, pages 258–269, 2002.

[25] B. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears. Benchmarking Cloud Serving Systems with YCSB. In *Proceedings of the 2010 ACM Symposium on Cloud Computing (SoCC '10)*, Indianapolis, Indiana, June 2010.

[26] B. Daniel, D. Dig, K. Garcia, and D. Marinov. Automated Testing of Refactoring Engines. In *Proceedings of the ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE '07)*, Dubrovnik, Croatia, September 2007.

[27] S. Dawson, F. Jahanian, and T. Mitton. Experiments on Six Commercial TCP Implementations Using a Software Fault Injection Tool. 27:1385–1410, 1997.

[28] S. Dawson, F. Jahanian, and T. Mitton. Experiments on Six Commercial TCP Implementations Using a Software Fault Injection Tool. *Software Practice and Experience*, 1997.

[29] J. Dean. Underneath the covers at google: Current systems and future directions. In *Google I/O*, San Francisco, California, May 2008.

[30] R. Deline and M. Fhndrich. Typestates for objects. In *European Conference on Object-Oriented Programming (ECOOP)*, 2004.

[31] C. Demartini, R. Iosif, and R. Sisto. A deadlock detection tool for concurrent java programs. *Software - Practice and Experience*, 29(7):577–603, 1999.

[32] A. Dinning and E. Schonberg. Detecting access anomalies in programs with critical sections. In *Proc. of the ACM/ONR Workshop on Parallel and Distributed Debugging*, 1991.

[33] O. Edelstein, E. Farchi, Y. Nir, G. Ratsaby, , and S. Ur. Multithreaded Java program test generation. *IBM Systems Journal*, 41(1):111–125, 2002.

[34] D. Engler and K. Ashcraft. RacerX: effective, static detection of race conditions and deadlocks. In *SOSP*, pages 237–252, 2003.

[35] D. R. Engler and K. Ashcraft. Racerx: effective, static detection of race conditions and deadlocks. In *19th ACM Symposium on Operating Systems Principles (SOSP)*, pages 237–252, 2003.

[36] E. Farchi, Y. Nir, and S. Ur. Concurrent bug patterns and how to test them. In *Proceedings of the 17th International Symposium on Parallel and Distributed Processing*, 2003.

[37] C. J. Fidge. Timestamp in message passing systems that preserves partial ordering. In *Proceedings of the 11th Australian Computing Conference*, pages 56–66, 1988.

[38] S. J. Fink, E. Yahav, N. Dor, G. Ramalingam, and E. Geay. Effective typestate verification in the presence of aliasing. In *International Symposium on Software Testing and Analysis (ISSTA)*, 2006.

[39] C. Flanagan and S. N. Freund. Atomizer: a dynamic atomicity checker for multi-threaded programs. In *31st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, pages 256–267, 2004.

[40] C. Flanagan and P. Godefroid. Dynamic partial-order reduction for model checking software. In *Proc. of the 32nd Symposium on Principles of Programming Languages (POPL'05)*, pages 110–121, 2005.

[41] C. Flanagan and P. Godefroid. Dynamic partial-order reduction for model checking software. In *Proceedings of the 32nd ACM SIGPLAN-SIGACT symposium on Principles of Programming Languages*, pages 110–121, 2005.

[42] C. Flanagan, K. Leino, M. Lillibridge, G. Nelson, J. B. Saxe, and R. Stata. Extended static checking for Java. In *ACM SIGPLAN conference on Programming Language Design and Implementation*, pages 234–245, 2002.

[43] D. Ford, F. Labelle, F. I. Popovici, M. Stokely, V.-A. Truong, L. Barroso, C. Grimes, and S. Quinlna. Availability in Globally Distributed Storage Systems. In *Proceedings of the 9th Symposium on Operating Systems Design and Implementation (OSDI '10)*, Vancouver, Canada, October 2010.

[44] C. Fu, B. G. Ryder, A. Milanova, and D. Wonnacott. Testing of Java Web Services for Robustness. In *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA '04)*, Boston, Massachusetts, July 2004.

[45] Q. Gao, W. Zhang, Z. Chen, M. Zheng, and F. Qin. 2ndstrike: Toward manifesting hidden concurrency typestate bugs. In *Architectural Support for Programming Languages and Operating Systems (ASPLOS'08)*. ACM press, 2011.

[46] G. Gibson. Reliability/Resilience Panel. In *High-End Computing File Systems and I/O Workshop (HEC FSIO '10)*, Arlington, VA, Aug. 2010.

[47] M. Gligoric, T. Gvero, V. Jagannath, S. Khurshid, V. Kuncak, and D. Marinov. Test generation through programming in UDITA. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering (ICSE '10)*, pages 225–234, Cape Town, South Africa, May 2010.

[48] P. Godefroid. Partial-order methods for the verification of concurrent systems - an approach to the state-explosion problem. volume 1032 of *Lecture Notes in Computer Science*. Springer-Verlag, 1996.

[49] P. Godefroid. Model checking for programming languages using verisoft. In *24th Symposium on Principles of Programming Languages*, pages 174–186, 1997.

[50] H. S. Gunawi, T. Do, P. Joshi, P. Alvaro, J. M. Hellerstein, A. C. Arpaci-Dusseau, R. H. Arpaci-Dusseau, and K. Sen. FATE and DESTINI: A Framework for Cloud Recovery Testing. In *Proceedings of the 8th Symposium on Networked Systems Design and Implementation (NSDI'11)*, Boston, Massachusetts, Mar. 2011.

[51] H. Guo, M. Wu, L. Zhou, G. Hu, J. Yang, and L. Zhang. Practical software model checking via dynamic interface reduction. In *SOSP*, 2011.

[52] J. Harrow. Runtime checking of multithreaded applications with visual threads. In *SPIN*, 2000.

[53] K. Havelund. Using runtime analysis to guide model checking of java programs. In *SPIN*, 2000.

[54] K. Havelund and T. Pressburger. Model checking java programs using java pathfinder. *Intl. Journal on Software Tools for Technology Transfer*, 2(4):366–381, 2000.

[55] A. Henry. Cloud Storage FUD: Failure and Uncertainty and Durability. In *Proceedings of the 7th USENIX Symposium on File and Storage Technologies (FAST '09)*, San Francisco, California, Feb. 2009.

[56] W. Hoarau, S. Tixeuil, and F. Vauchelles. FAIL-FCI: Versatile Fault Injection. Journal of Future Generation Computer Systems, Volume 23 Issue 7, August, 2007.

[57] C. A. R. Hoare. Monitors: an operating system structuring concept. *Communications of the ACM*, pages 549–557, 1974.

[58] T. Hoff. Netflix: Continually Test by Failing Servers with Chaos Monkey. `http://highscalability.com`, Dec. 2010.

[59] G. Holzmann. The Spin model checker. *IEEE Transactions on Software Engineering*, 23(5):279–295, 1997.

[60] D. Hovemeyer and W. Pugh. Finding concurrency bugs in java. In *Proceedings of the PODC Workshop on Concurrency and Synchronization in Java Programs*, 2004.

[61] P. Hunt, M. Konar, F. P. Junqueira, and B. Reed. ZooKeeper: Wait-free coordination for Internet-scale systems. In *USENIX ATC '10*.

[62] A. Johansson and N. Suri. Error Propagation Profiling of Operating Systems . In *Proceedings of the International Conference on Dependable Systems and Networks (DSN '05)*, Yokohama, Japan, June 2005.

[63] P. Joshi, H. S. Gunawi, and K. Sen. PreFail: A Programmable Tool for Multiple-Failure Injection. In *ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications.*, 2011.

[64] P. Joshi, H. S. Gunawi, and K. Sen. PreFail: A Programmable Failure-Injection Framework. UC Berkeley Technical Report UCB/EECS-2011-30, Apr. 2011.

[65] P. Joshi, M. Naik, C.-S. Park, and K. Sen. An extensible active testing framework for concurrent programs. In *International Conference on Computer Aided Verification*, Lecture Notes in Computer Science. Springer, 2009.

[66] P. Joshi, M. Naik, K. Sen, and D. Gay. An effective dynamic analysis for detecting generalized deadlocks. In *ACM SIGSOFT Symposium on the Foundations of Software Engineering*, 2010.

[67] P. Joshi, C.-S. Park, K. Sen, and M. Naik. A randomized dynamic program analysis technique for detecting real deadlocks. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2009.

[68] P. Joshi and K. Sen. Predictive typestate checking of multithreaded java programs. In *IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2008.

[69] H. Jula, D. Tralamazza, C. Zamfir, and G. Candea. Deadlock immunity: Enabling systems to defend against deadlocks. In *OSDI*, 2008.

[70] L. Juszczyk and S. Dustdar. Programmable Fault Injection Testbeds for Complex SOA. In *Proceedings of the 8th International Conference on Service Oriented Computing (ICSOC '10)*, San Francisco, California, December 2010.

[71] L. Keller, P. Marinescu, and G. Candea. AFEX: An Automated Fault Explorer for Faster System Testing. 2008.

[72] N. E. B. Kevin Bierhoff and J. Aldrich. Practical api protocol checking with access permissions. In *European Conference on Object-Oriented Programming (ECOOP)*, 2009.

[73] C. Killian, J. W. Anderson, R. Braud, R. Jhala, and A. Vahdat. Mace: Language support for building distributed systems. In *PLDI'07*.

[74] P. Koopman and J. DeVale. Comparing the Robustness of POSIX Operating Systems. In *Proceedings of the 29th International Symposium on Fault-Tolerant Computing (FTCS-29)*, Madison, Wisconsin, June 1999.

[75] A. Lakshman and P. Malik. Cassandra - A Decentralized Structured Storage System. In *The 3rd ACM SIGOPS International Workshop on Large Scale Distributed Systems and Middleware (LADIS '09)*.

[76] T. Li, C. S. Ellis, A. R. Lebeck, and D. J. Sorin. Pulse: A dynamic deadlock detection mechanism using speculative execution. In *Proceedings of the 2005 USENIX Annual Technical Conference*, 2005.

[77] P. Marinescu and G. Candea. LFI: A Practical and General Library-Level Fault Injector. In *DSN '09*.

[78] P. D. Marinescu, R. Banabic, and G. Candea. An Extensible Technique for High-Precision Testing of Recovery Code. In *Proceedings of the USENIX Annual Technical Conference (ATC '10)*, Boston, Massachusetts, June 2010.

[79] S. Masticola. *Static detection of deadlocks in polynomial time*. PhD thesis, Rutgers University, 1993.

[80] S. Masticola and B. Ryder. A model of Ada programs for static deadlock detection in polynomial time. In *Proceedings of ACM/ONR Workshop on Parallel and Distributed Debugging*, pages 97–107, 1991.

[81] F. Mattern. Virtual time and global states of distributed systems. In *Proceedings of the Parallel and Distributed Algorithms Conference*, Elsevier Science, pages 215–226, 1988.

[82] F. Mattern. Virtual time and global states of distributed systems. In *Proceedings of Workshop on Parallel and Distributed Algorithms*, pages 215–226, North-Holland / Elsevier, 1989. (Reprinted in: Z. Yang, T.A. Marsland (Eds.), "Global States and Time in Distributed Systems", IEEE, 1994, pp. 123-133.).

[83] A. Milanova, A. Rountev, and B. Ryder. Parameterized object sensitivity for points-to analysis for Java. *ACM Transactions on Software Engineering and Methodology*, 14(1):1–41, Jan. 2005.

[84] M. Naik, C.-S. Park, K. Sen, and D. Gay. Effective static deadlock detection. In *ICSE*, 2009.

[85] Y. Nir-Buchbinder, R. Tzoref, and S. Ur. Deadlocks: From exhibiting to healing. In *8th Workshop on Runtime Verification*, 2008.

[86] R. O'Callahan and J.-D. Choi. Hybrid dynamic data race detection. In *ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 167–178. ACM, 2003.

[87] C.-S. Park and K. Sen. Randomized active atomicity violation detection in concurrent programs. In *16th International Symposium on Foundations of Software Engineering (FSE'08)*. ACM, 2008.

[88] D. Patterson, G. Gibson, and R. Katz. A Case for Redundant Arrays of Inexpensive Disks (RAID). In *Proceedings of the 1988 ACM SIGMOD Conference on the Management of Data (SIGMOD '88)*, Chicago, Illinois, June 1988.

[89] E. Pinheiro, W.-D. Weber, and L. A. Barroso. Failure Trends in a Large Disk Drive Population. In *Proceedings of the 5th USENIX Symposium on File and Storage Technologies (FAST '07)*, San Jose, California, February 2007.

[90] V. Prabhakaran, L. N. Bairavasundaram, N. Agrawal, H. S. Gunawi, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. IRON File Systems. In *Proceedings of the 20th ACM Symposium on Operating Systems Principles (SOSP '05)*, Brighton, United Kingdom, October 2005.

[91] C. J. Price and N. S. Taylor. Automated multiple failure FMEA. *Reliability Engineering and System Safety*, 76(1):1–10, Apr. 2002.

[92] A. V. Raman and J. D. Patrick. The sk-strings method for inferring pfsa. In *Proceedings of the workshop on automata induction, grammatical inference and language acquisition at the 14th international conference on machine learning (ICML97)*, 1997.

[93] 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*, 15(4):391–411, 1997.

[94] B. Schroeder and G. Gibson. Disk failures in the real world: What does an MTTF of 1,000,000 hours mean to you? In *Proceedings of the 5th USENIX Symposium on File and Storage Technologies (FAST '07)*, San Jose, California, Feb. 2007.

[95] A. Sen and V. K. Garg. Partial order trace analyzer (pota) for distributed programs. In *Proceedings of the 3rd Workshop on Runtime Verification (RV03)*, Electronic Notes in Theoretical Computer Science, 2003.

[96] K. Sen. Race directed randomized dynamic analysis of concurrent programs. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'08)*, 2008.

[97] K. Sen, G. Rosu, and G. Agha. Runtime safety analysis of multithreaded programs. In *Proceedings of 4th joint European Software Engineering Conference and ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE 03)*, ACM, 2003.

[98] K. Sen, G. Rosu, and G. Agha. Online efficient predictive safety analysis of multithreaded programs. In *10th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'04)*, volume 2988 of *Lecture Notes in Computer Science*, pages 123–138, 2004.

[99] S. Shoham, E. Yahav, S. Fink, and M. Pistoia. Static specification mining using automata-based abstractions. In *International Symposium on Software Testing and Analysis*, 2007.

[100] K. Shvachko, H. Kuang, S. Radia, and R. Chansler. The Hadoop Distributed File System. In *Proceedings of the 26th IEEE Symposium on Massive Storage Systems and Technologies (MSST '10)*.

[101] J. Simsa, R. Bryant, and G. Gibson. dbug: Systematic evaluation of distributed systems. In *Proceedings of the 5th International Workshop on Systems Software Verification*, 2010.

[102] Soot: A java optimization framework. `http://www.sable.mcgill.ca/soot/`.

[103] S. D. Stoller. Testing concurrent Java programs using randomized scheduling. In *Workshop on Runtime Verification (RV'02)*, volume 70 of *ENTCS*, 2002.

[104] R. E. Strom and S. Yemini. Typestate: A programming language concept for enhancing software reliability. *IEEE Transactions on Software Engineering*, 12(1):157–171, 1986.

[105] H. Team. Hadoop Fault Injection Framework and Development Guide. `http://hadoop.apache.org/hdfs/docs/r0.21.0/faultinject_framework.html`.

[106] K. Vishwanath and N. Nagappan. Characterizing Cloud Computing Hardware Reliability. In *Proceedings of the 2010 ACM Symposium on Cloud Computing (SoCC '10)*, Indianapolis, Indiana, June 2010.

[107] C. von Praun. *Detecting Synchronization Defects in Multi-Threaded Object-Oriented Programs*. PhD thesis, Swiss Federal Institute of Technology, Zurich, 2004.

[108] C. von Praun and T. R. Gross. Object race detection. In *16th ACM SIGPLAN conference on Object oriented programming, systems, languages, and applications (OOPSLA)*, pages 70–82. ACM, 2001.

[109] L. Wang and S. D. Stoller. Run-time analysis for atomicity. In *3rd Workshop on Run-time Verification (RV'03)*, volume 89 of *ENTCS*, 2003.

[110] L. Wang and S. D. Stoller. Accurate and efficient runtime detection of atomicity errors in concurrent programs. In *Proc. ACM SIGPLAN 2006 Symposium on Principles and Practice of Parallel Programming (PPoPP)*, pages 137–146. ACM Press, Mar. 2006.

[111] T. White. *Hadoop The Definitive Guide*. O'Reilly, 2009.

[112] A. Williams, W. Thies, and M. Ernst. Static deadlock detection for Java libraries. In *ECOOP*, 2005.

[113] B. Xin, W. N. Sumner, and X. Zhang. Efficient program execution indexing. In *ACM SIGPLAN conference on Programming language design and implementation*, pages 238–248, 2008.

[114] J. Yang, T. Chen, M. Wu, Z. Xu, X. Liu, H. Lin, M. Yang, F. Long, L. Zhang, and L. Zhou. MODIST: Transparent Model Checking of Unmodified Distributed Systems. In *NSDI '09*.

[115] J. Yang, P. Twohey, D. Engler, and M. Musuvathi. Using Model Checking to Find Serious File System Errors. In *OSDI'04*.

[116] Y. Yang, A. Gringauze, D. Wu, and H. Rohde. Detecting data race and atomicity violation via typestate-guided static analysis. Technical Report MSR-TR-2008-108, Microsoft Research, 2008.