TESTING, RUNTIME VERIFICATION, AND ANALYSIS OF CONCURRENT PROGRAMS

BY

QINGZHOU LUO

DISSERTATION

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy in Computer Science
in the Graduate College of the
University of Illinois at Urbana-Champaign, 2015

Urbana, Illinois

Doctoral Committee:

Professor Grigore Roşu, Chair
Professor Darko Marinov
Professor Tao Xie
Dr. Klaus Havelund, NASA Jet Propulsion Laboratory

Abstract

With the development of multi-core processors, concurrent programs are becoming more and more popular. Among several models, the multithreaded shared-memory model is the predominant programming paradigm for developing concurrent programs. However, because of non-deterministic scheduling, multithreaded code is hard to develop and test. Concurrency bugs, such as data races, atomicity violations, and deadlocks, are hard to detect and fix in multithreaded programs.

To test and verify multithreaded programs, two sets of techniques are needed. The first one is to *enforce* thread schedules and runtime properties efficiently. Being able to enforce desired thread schedules and runtime properties would greatly help developers to develop reliable multithreaded code. The second one is to *explore* the state space of multithreaded programs efficiently. Systematic state-space exploration could guarantee correctness for multithreaded code, however, it is usually time consuming and thus infeasible in most cases.

This dissertation presents several techniques to address challenges arising in testing and runtime verification of multithreaded programs. The first two techniques are the IMUnit framework for enforcing testing schedules and the EnforceMOP system for enforcing runtime properties for multithreaded programs. An experimental evaluation shows that our techniques can enforce thread schedules and runtime properties effectively and efficiently, and have their own advantages over existing techniques. The other techniques are the RV-Causal framework and the CAPP technique in the ReEx framework for efficient state-space exploration of multithreaded code. RV-Causal employs the idea of the maximal causal model for state-space exploration in a novel way to reduce the exploration cost, without

losing the ability to detect certain types of concurrency bugs. The results show that RV-Causal outperforms existing techniques by finding concurrency bugs and exploring the entire state space much more efficiently.

To my family

Acknowledgments

Nothing would be possible for me to achieve without the enormous support from the people around me. I want to take this chance to express my gratitude to all the people, although I know I can never thank them enough.

Specifically, I would like to thank:

- Prof. Darko Marinov for his guidance, advice and support through my PhD study. Darko taught me so many basic things at the very beginning, and I will never forget the most important lessons I have learned from Darko: stay curious, care for details and work hard.

- Prof. Grigore Roşu for teaching me how to look at things by their nature rather than their appearance. I also want to thank Grigore for letting me know how to stay focused on one thing and do solid work.

- Prof. Tao Xie and Dr. Klaus Havelund for their great advice and help for serving on my PhD committee.

- Prof. Danny Dig for his wise advice and collaboration.

- My colleagues from the Mir group, including Milos Gligoric, Vilas Jagannath, Adrian Nistor, Yu Lin, Farah Hariri, and Lamyaa Eloussi, for their vital collaboration during my PhD study.

- My colleagues from the FSL group, including Yi Zhang, Choonghwan Lee, Dongyun

Jin, Jeff Huang, and Brandon Moore for their collaboration and fun moments we spent together.

- John Micco for his great help during my two internships at Google. John always answered my questions instantly, and our flaky tests study would be impossible without the help and mentoring from John.
- Dr. Mihai Budiu for his feedback and guidance during my internship at MSR.
- My friends at UIUC, including Peiwen Wu, Xinying Zong, Hanze Ying, Yanfeng Zhang, Kai-wei Chang and many others, for keeping my five years colorful.
- My friends at Champaign Table Tennis Club, including Bryant, Robert, Junmei, David, Nianhua and many others, for keeping me healthy and happy.
- My parents for their unconditional support at anytime and anywhere, and my wife Qian for her unbounded support, tolerance and love for me. Last but not least, my daughter Evelyn for the unlimited happiness she brought to me in the last year of my PhD study.

I apologize to anyone that I may have inadvertently omitted and thank them for being a part of the journey of my PhD study.

Table of Contents

Chapter 1

Introduction

Concurrent programs are getting more and more popular with the advancement of multi-core processors. To extract greater performance from multi-core processors, developers need to write parallel code, either from scratch or by transforming sequential code. The predominant paradigm for writing parallel code is that of shared memory where multiple threads of control communicate by reading and writing shared data objects. Shared-memory multithreaded code is often afflicted by bugs such as data races, atomicity violations, and deadlocks. These bugs are hard to detect because multithreaded code can demonstrate different behavior based on the scheduling of threads, and the bugs may only be triggered by a small specific set of schedules. Hence it is challenging to build reliable multithreaded software.

Testing and runtime verification are two different ways to improve the reliability of software. A test for a program consists of test input, test code, and test oracles (test assertions). When a test is being executed, its test oracle will decide whether the test passes or not. A test failure usually indicates that there are bugs inside the system under test (SUT). Runtime verification takes a different approach from testing to improve software reliability. Runtime verification takes user defined events and properties (which are usually temporal orders between events) as input, and checks whether those properties hold or not at runtime. Whenever a property violation is found, recovery code can be executed, which can be any code, in particular a warning being displayed to the user.

While both testing and runtime verification are known to be effective for sequential programs, applying them for concurrent programs faces some new challenges. Because of the non-deterministic scheduling of multithreaded code, the first challenge is how to *enforce* cer-

tain schedules when testing or verifying multithreaded programs. For testing, a deterministic schedule is needed because it would help in validating the test oracles. For runtime verification, since there are multiple possible thread schedules for a given multithreaded program, the fact that no violation is detected in one particular schedule does not guarantee that the desired property will hold in all possible schedules. The second challenge is how to efficiently *explore* all the possible thread schedules. Testing or runtime verification cannot cover all the possible thread schedules that a given multithreaded program can manifest, therefore techniques for exploring all the possible thread schedules are needed to guarantee correctness. Existing techniques enumerate *all* the possible thread schedules for a given multithreaded program; however this task is usually very time consuming and thus infeasible in most cases.

In this dissertation we aim to address the above issues of testing and runtime verification for multithreaded programs. First, we propose techniques for expressing and enforcing schedules in multithreaded tests, as well as an extension to the JavaMOP runtime verification framework to enforce formal properties for multithreaded programs. Second, we also propose techniques for efficient state-space exploration to find concurrency bugs in multithreaded programs. Evaluations show that our proposed techniques provide efficient ways to enforce testing schedules and runtime properties, and also make state-space exploration for multithreaded programs much more efficient compared to existing techniques.

1.1 Thesis Statement

Our thesis statement is the following:

> *It is possible to improve the reliability of multithreaded programs by (1) effectively enforcing correct thread schedules and temporal properties and (2) efficiently exploring interleavings that could possibly trigger concurrency bugs.*

To confirm this statement, this dissertation presents two main bodies of research organized as follows: (1) the IMUnit framework for writing multithreaded tests and the EnforceMOP system for enforcing any general temporal properties for multithreaded programs, and (2) the CAPP technique for change-aware exploration of multithreaded tests and the RV-Causal framework for exploration reduction based on the maximal causal model.

1.2 Enforcement of Testing Schedules and Runtime Properties

1.2.1 Enforcing Schedules for Multithreaded Tests

To validate multithreaded code, developers write multithreaded unit tests. A multithreaded test creates and executes two or more threads (and/or invokes code under test that itself creates and executes two or more threads). Each test execution follows some schedule/interleaving of the multiple threads, and different schedules can give different results. Developers often want to enforce a particular schedule for a test. For example, consider having two threads, one that executes a method m, and the other that executes a method m'. Developers may want to ensure in one test that m finishes before m' starts on the other thread and in another test that m' finishes before m starts (and in more tests that m and m' interleave in certain ways). Without controlling the schedule, it is impossible to write precise assertions

about the execution because the results can differ in the two scenarios, and it is impossible to guarantee which scenarios were covered during testing, even if multiple testing runs are performed.

To control the schedule of multithreaded tests, developers mostly use a combination of timed delays in the various test threads. In Java, the delay is performed with the `Thread.sleep` method, so we call this approach *sleep-based*. A sleep pauses a thread while other threads continue execution. Using a combination of sleeps, developers attempt to enforce the desired schedule on the execution of a multithreaded test, and then assert the intended result for the desired schedule. A sleep-based test can fail when an undesired schedule gets executed even if the code under test has no bug (false positive). Dually, a sleep-based test can pass when an unintended schedule gets executed even if the code under test has a bug (false negative). Indeed, sleeps are an unreliable and inefficient mechanism for enforcing schedules because sleeps are based on real time. To use sleeps, one has to estimate the real-time duration for which to delay a thread while the other threads perform their work. This is usually estimated by trial and error, starting from a small duration and increasing it until the test passes consistently on the developer's machine. The estimated duration depends on the execution environment (hardware/software configuration and the load on the machine) on which the delay time is being estimated. Therefore, when the same test is executed in a different environment, the intended schedule may not be enforced leading to false positives/negatives. Moreover, sleep can be very inaccurate even on a single machine [57]. In an attempt to mitigate the unreliability of sleep, developers often end up over-estimating the duration, which in turn leads to slow running multithreaded tests.

We present a new framework, called *IMUnit*, which aims to address these issues with multithreaded unit testing. IMUnit introduces a novel language that enables natural and explicit specification of schedules for multithreaded unit tests. Semantically, the basic entity in an IMUnit schedule is an *event* that an execution can produce at various points (e.g., a thread starting/finishing the execution of a method, or a thread getting blocked). We call the

IMUnit approach *event-based*. An IMUnit schedule itself is a (monitorable) property [23,65] on the sequence of events. More precisely, each schedule is expressed as a set of desirable *event orderings*, where each event ordering specifies the order between a pair of events (note that an IMUnit schedule need not specify a total order between all events but only the necessary partial order).

1.2.2 Runtime Property Enforcement for Concurrent Programs

Other than testing, runtime verification combines formal methods and testing to check critical properties of a program dynamically. The key idea is that software system properties, often defined using temporal formalisms, can be used to generate program monitors. Any property violation is reported or resolved immediately rather than waiting for a bug to manifest. Runtime verification has been proven to be a promising technique to increase software reliability, with a large number of runtime verification techniques and tools developed, including Hawk/Eraser [27], MOP [22], PQL [64], PTQL [38] and Tracematch [1], among many others.

While runtime verification can effectively detect property violations, and sometimes even recover from such violations, unfortunately it provides no guarantee that properties are never violated. This is particularly problematic in multithreaded systems, where non-deterministic thread scheduling may hide potentially critical errors. For example, consider a concurrent database where one thread is in charge of authorizing users, and each user is assigned a thread for fetching data. The underlying property is that any user should be authorized before getting data, so for any given user the corresponding thread should wait until the first thread finishes authorizing. Runtime verification approaches can monitor the program execution and report violations of this property for each user, but *cannot* prove correctness: a successful run gives no guarantee that other runs, under different thread schedules, will also be successful.

The conventional approach is to employ language-specific synchronization mechanisms or

ad-hoc sleep commands to enforce such properties when developing or testing multithreaded programs. For instance, Java provides a `synchronized` keyword, a `Thread.sleep()` method, and several other classes in the `java.util.concurrent` package. However, there are certain limitations when using these constructs to enforce arbitrary properties in multithreaded programs: (1) it is non-trivial and error-prone to use these constructs when the property to be enforced is complex, as shown later in this dissertation; and (2) all these constructs are mingled with the original program. Therefore, these constructs are not modular, and they also make it hard to identify and reason about the underlying properties that the developers are attempting to enforce.

Here we present EnforceMOP, a novel framework for enforcing complex properties in multithreaded programs. The properties are enforced at runtime and do not require to modify the source code, so they can be modularly maintained. We show that EnforceMOP can be used effectively both in developing and in testing multithreaded programs.

We make the following specific contributions:

Technique: We propose a technique to enforce arbitrarily complex safety properties in multithreaded programs. The properties can be expressed using various formalisms.

Implementation: EnforceMOP is implemented in Java on top of JavaMOP [23], a state-of-the-art runtime verification framework. Following the philosophy of JavaMOP, EnforceMOP is implemented in a logic-independent way.

Evaluation: We evaluated the effectiveness of EnforceMOP in two aspects. First, as a framework to enforce general properties when developing multithreaded programs, specifically to enforce correct behaviors of such programs. Second, as a testing framework to enforce thread schedules when unit testing multithreaded programs, specifically to enforce schedules in 185 existing multithreaded unit tests, and compared it with several existing testing frameworks.

1.3 Efficient State-Space Exploration

1.3.1 Efficient Exploration of Multithreaded Regression Tests

Ensuring the reliability of multithreaded code has been an active area of research with several promising recent results [16, 17, 26, 33, 34, 52, 67, 70, 75]. Most of these tools execute multithreaded tests and check for the presence of faults. Since multithreaded code can have different behavior for different thread schedules/interleavings, these tools conceptually *explore* the code for a large number of schedules. As a result, the tools typically take fairly long time to check code. Moreover, most existing tools are *change-unaware*: they check only one version of code at a time, and do not exploit the fact that code evolves over several versions during development and maintenance.

Regression testing involves re-executing the tests for a program when its code changes to ensure that the changes have not introduced a fault that causes test failures. As programs evolve and grow, their test suites also grow, and over time it becomes expensive to re-execute all the tests. The problem is exacerbated when test suites contain multithreaded tests that are generally long running. While many techniques have been proposed to alleviate this problem for sequential tests [90], there is much less work for multithreaded code [36, 89].

We propose a novel technique, called *Change-Aware Preemption Prioritization (CAPP)*, that uses information about the changes in software evolution to prioritize the exploration of schedules in a multithreaded regression test. The goal of CAPP is to find a fault faster, if one exists. Our technique decides in what order to explore thread schedules based on how test exploration dynamically encounters changed code.

CAPP is a general technique that can be instantiated with different definitions of code changes and types of ordering for schedules. We present 14 heuristics that consider changes at the level of source-code lines/statements, methods, classes, or fields affected by the change, and that consider orderings based on whether all or only some executing threads are encountering changed code.

1.3.2 Systematic Testing of Concurrent Programs with Maximal Causality

A fundamental challenge in testing concurrent programs is how to effectively cover the astronomical thread interleaving or scheduling space to either find out the buggy thread interleaving or prove the program is correct. In theory, a bug may be hidden anywhere in the state space and finding it is as hard as finding a needle in a haystack. Worse, the diversity of the exercised interleavings tends to be highly correlated with the execution environments [69,91]. Naively executing the program on the same platform repeatedly (such as stress testing) results in redundant exploration of similar interleavings, keeping the buggy interleaving space still uncovered.

Systematic testing approaches [18, 66, 68, 69, 88, 91] offer a more promising solution for testing concurrent programs. They avoid testing repeated interleavings by actively controlling the thread scheduler to systematically explore all legal but distinct interleavings. If a buggy interleaving is hit during the exploration, then that interleaving can be used to reproduce the bug. If no buggy interleaving is found after the exploration finishes, then the concurrent program is proven to be correct. Systematic testing is more effective than straightforwardly repeated execution of the concurrent program because it guarantees that each test execution covers a different interleaving. However, the core challenge still remains: to cover the astronomical scheduling space, the same astronomical number of test executions must be done.

Researchers have proposed various methods to reduce the exploration space for systematic testing approaches. For example, context bounding techniques [66,68] limit the number of preemptions each explored interleaving could have, and coverage-driven techniques [88,91] and priority-based techniques [18,48,69] prioritize schedules during exploration. Those techniques have been proven to be effective for finding certain bugs in concurrent programs. However, they only try to select or prioritize schedules in exploration space, so those techniques

cannot prove the program is correct and may also miss bugs.

Several researchers have also proposed partial-order reduction techniques [35, 37] to reduce the cost of state-space exploration. The idea is to only explore schedules that lead to different program states. For example, dynamic partial-order reduction (DPOR) techniques [35] prune state space by looking at all the currently active transitions and only explore one of them if they do not interfere with one another. DPOR can indeed greatly reduce state-space exploration cost. However, it is based on *happens-before* causality, and thus, it does not prune the maximal possible number of interleavings. In other words, many interleavings explored by DPOR could possibly belong to the same maximal causal model [82].

We propose a new approach that systematically explores the interleaving space with substantially fewer number of test executions. Our key insight is to look at the state-space exploration problem from the perspective of the *causal model* (instead of interleavings), which characterizes a set of legal interleavings that are causally equivalent and can be derived from one another. These causal sets have the important property that if any single interleaving is tested then there is no need to test any other interleaving in the same causal set. Moreover, the checking of each causal set can be done offline and in parallel, so our technique is particularly suitable when online testing is more expensive than offline checking.

Generally speaking, the classical happens-before relationship [56] (HB) yields such a causal model. HB characterizes the set of interleavings in which the order between operations can be altered if they have no HB relation. However, HB is rather strict, in that its power of characterizing causality is quite limited. Instead, our approach builds upon the *maximal causal model* [82] (MCM) technique, which yields the largest possible causal equivalence classes. In other words, from any single execution trace, MCM is able to derive the largest causal set of legal interleavings w.r.t. that trace. Underpinned by this property, our approach minimizes the number of executions that are needed to run to cover the entire scheduling space w.r.t. a given input.

A main technical challenge here is how to systematically generate new schedules such

that: (1) no two subspaces (corresponding to two different traces) overlap, and (2) all the subspaces together cover the entire scheduling space. Our approach works by pivoting around the value of reads in the trace. Specifically, we ensure that each generated new schedule has at least one new event: a read event that reads a new data (i.e., a different value from that in other schedules). All such new events are considered and their corresponding schedules are generated, as long as the schedule is legal (permitted by MCM). In this way, we guarantee that no two schedules are redundant in terms of MCM, i.e., the corresponding trace of each schedule contains at least one distinct event compared to others. Moreover, because the generated schedules consider all possible legal combinations of read values, our approach would cover the entire state space eventually.

We make the following specific contributions:

- A new systematic concurrency testing approach that leverages the maximal causal model to minimize the number of executions needed to execute, and shifts the runtime computation cost to offline inference and property checking through constraint solving.
- A schedule generation technique that systematically generates new schedules that cover distinct thread interleavings until the whole scheduling space is covered.
- A set of evaluations that shows that our technique is able to find concurrency bugs and explore the entire state space more efficiently and effectively than the existing techniques.

1.4 Dissertation Organization

The rest of this dissertation is organized as follows:

- Chapter 2 presents a brief overview of background techniques used in this dissertation, such as the JavaMOP framework, the ReEx framework, and the Maximal Causal Model. It serves as the foundation for the rest of this dissertation.

- Chapter 3 presents our enforcement approach for executing multithreaded programs and tests. It describes the IMUnit framework and the EnforceMOP system, a runtime system for enforcing any temporal properties in multithreaded programs.

- Chapter 4 presents our exploration approach for testing and verifying multithreaded programs. We first briefly present the CAPP framework, which prioritizes thread schedules based on the changes between two versions. We then present the RV-Causal framework in detail for efficient state-space exploration based on the maximal causal model.

- Chapter 5 presents an overview of research related to this dissertation.

- Chapter 6 concludes this dissertation, followed by various possible future work directions based on this work.

Chapter 2

Background

This chapter presents the technical background for techniques presented in this dissertation. Section 2.1 presents two different approaches for testing multithreaded programs: enforcing the correct thread schedules and exploring the entire state space. Section 2.2 introduces the runtime verification techniques and explains how they apply to multithreaded programs.

2.1 Testing and Exploration of Multithreaded Programs

With the advent of multi-core processors, concurrent programs become more and more popular in practice. Concurrent programs provide an efficient yet convenient way to do multiple tasks at the same time. Among others, the *multithreaded, shared-memory* paradigm is the dominating programming paradigm so far. A multithreaded program consists of several threads being executed at the same time. All the threads can access the same memory region of the process, so they can read or modify shared variables concurrently. Although this model is convenient for programmers to write multithreaded code, it also introduces *non-determinism* in the program caused by the different execution orders between threads. Such non-determinism is very hard to test, reproduce and fix. Here we use an example to demonstrate it.

In Figure 2.1, two threads are started concurrently in the program. They both access the same field `value` of a same object. Depending on the execution order of those two threads,

```
1  class Counter {
2
3      public int value = 0;
4
5      public static void main(String[] args) {
6
7          final Counter c = new Counter();
8
9          Thread t1 = new Thread(new Runnable() {
10                 @Override
11                 public void run() {
12                     int temp = c.value;
13                     temp++;
14                     c.value = temp;
15                 }
16             });
17
18         Thread t2 = new Thread(new Runnable() {
19                 @Override
20                 public void run() {
21                     int temp = c.value;
22                     temp++;
23                     c.value = temp;
24                 }
25             });
26
27         t1.start();
28         t2.start();
29
30         t1.join();
31         t2.join();
32
33         System.out.println(c.value);
34     }
35 }
```

Figure 2.1: Counter example

the result of `value` could vary. For example, if `t1` finishes execution before `t2` starts, then `value` will have the value `2`. In another execution, if `t2` starts to execute in the middle of the execution of `t1` on line 13 (which is called as a *context switch* or *preemption*), then `value` will have the value `1`. Therefore, different thread schedules will lead to different results of the program's execution.

There are several different approaches to addressing the non-determinism of multithreaded programs. The first one is to use some synchronization mechanisms to *enforce* the correct schedules that is being executed, thus only a subset among all the possible thread schedules can be executed. The second approach is to *systematically explore* all the possible thread schedules in the program. Any problematic schedules will be found during the exploration, therefore they can be fixed by developers. In the rest of this section we introduce the

background of both approaches.

2.1.1 Enforcing Correct Thread Schedules

One way to address the issue of non-determinism of multithreaded programs is to enforce only a subset thread schedules that can be executed. In practice, developers often achieve this goal by inserting `Thread.sleep` statements in the program to force a context switch at certain points. For example, developers can insert `Thread.sleep` before line 21 in Figure 2.1, right before the start of `t2`. By doing this, it is likely that `t1` will first finish execution before `t2` starts.

Enforcement of certain thread schedules can be used for both testing and development of multithreaded programs. When testing multithreaded programs, developers usually want to test a specific scenario, and the test oracle (`assertion`) depends on the thread schedules to be tested. When developing multithreaded programs, it is also convenient to be able to enforce certain schedules to avoid bugs or to improve performance.

However using `Thread.sleep` statements has its own drawbacks, such as being not intuitive, error prone and impacting performance. Researchers have developed various techniques to overcome those disadvantages of using `Thread.sleep`, such as ConAn [59] and MultithreadTC [78]. In Chapter 3 of this dissertation, we present the IMUnit framework for enforcing testing schedules, and the EnforceMOP system for enforcing both testing schedules and temporal properties at runtime. We also compare IMUnit and EnforceMOP with existing approaches.

2.1.2 Exploring State Space

Another way to address the issue of non-determinism in multithreaded programs is to systematically explore all the possible thread schedules. By doing this, the problematic thread schedules can be found and fixed by developers before they occur in practice. Exploration

of multithreaded programs works by incrementally executing different thread schedules until all the thread schedules have been enumerated. Since a context switch can occur at many different points in the program, all the possible thread schedules for a given program can be organized as a *exploration tree*, with the root being the start of the program and each edge representing a choice of a particular thread to execute at that given point. In Figure 2.2 we present the exploration tree for the program in Figure 2.1.

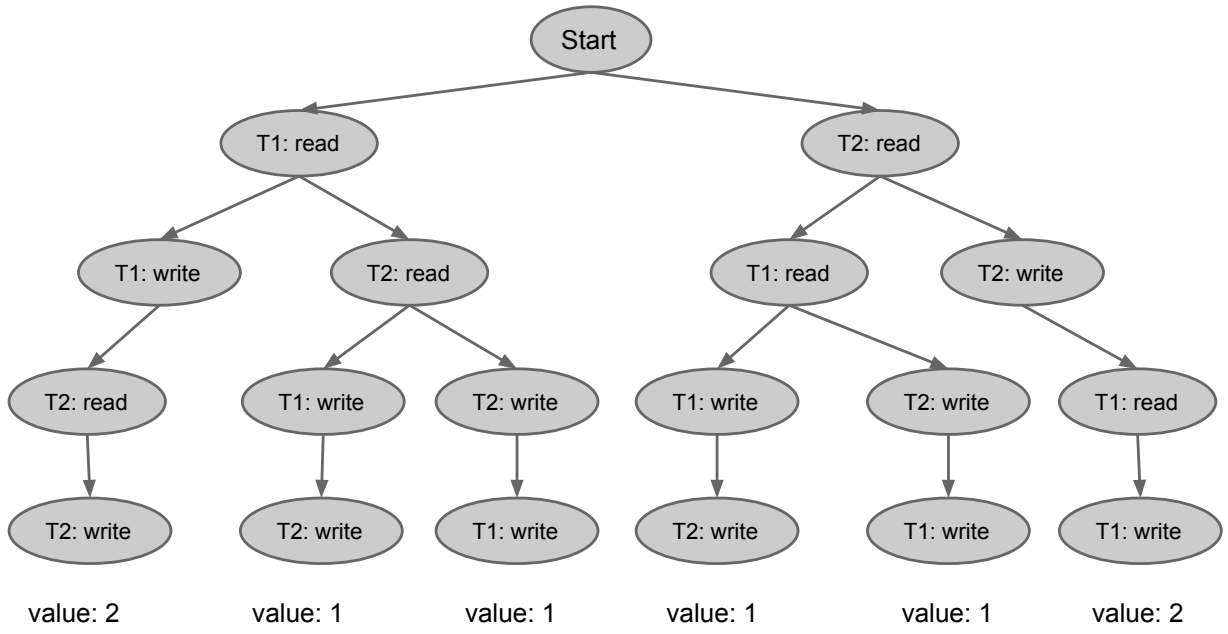


Figure 2.2: Exploration Tree

Each thread performs a read operation for the shared variable followed by a write operation. When the program starts, it can choose to run `t1` first or `t2` first. After the first read operation, it can also choose to continue executing the first thread with its write operation, or context switch to another thread. By enumerating all the possible choices in the programs' execution, we get a tree-like graph shown in Figure 2.2. Each leaf node represents a completed thread schedule for the program, with possibly different execution result.

Exploration can help discovering buggy thread schedules, but it also has the problem of state-space explosion. At each choice point there are multiple threads can be chosen from,

so the total number of thread schedules is exponential to the number of choice points in the program. There are different ways to explore the entire state space efficiently. The first one is called *stateful exploration*, which models the program state at each step and use *backtrack* and *state comparison* to explore new schedules. The advantage of this approach is that it can merge same states together, thus alleviating the problem of state space explosion. However it comes with the cost that it has to model and maintain the program state. Java PathFinder (JPF) [53] is the state-of-the-art stateful exploration tool for Java programs. It works on Java bytecode by modeling all the different kinds of bytecode instructions respectively.

Another approach (*stateless exploration*) is to do exploration without modeling the program states explicitly. Each schedule is recorded by keeping track of all the choices made during its execution. After finishing one execution, the program will re-execute from the very beginning, but mutate the recorded choice points to force the program to execute a new schedule. Compared to stateful exploration, stateless exploration does not need to model and maintain the program state, thus it can finish executing a single thread schedule faster. However, the total number of schedules may be much larger because of the duplicated program states are not discovered and merged. The ReEx framework is a stateless exploration tool developed by ourselves for Java programs. ReEx also works on Java bytecode, and it supports different exploration strategies to select or prioritize the schedules to be explored. The rest of this dissertation makes use of ReEx heavily to demonstrate and evaluate our techniques.

2.2 Runtime Verification of Multithreaded Programs

Runtime verification is another way to improve software reliability by combining formal methods with testing. It takes as input a set of user defined events and formal properties, and verifies those properties at *runtime*. Since runtime verification does not aim to statically verify all the possible hypothetical executions, it has the potential to be used more widely

in practice.

For multithreaded programs, runtime verification techniques will only verify the current thread schedule that is being executed. Therefore, it could potentially miss buggy schedules. In this section, we first introduce the state-of-the-art runtime verification framework JavaMOP [65]. Then we introduce the maximal causal model [82] and show how runtime verification techniques can help verifying multithreaded programs. Later on in this dissertation, we show how our techniques built upon JavaMOP and the maximal causal model can help enforcing and exploring thread schedules efficiently.

2.2.1 JavaMOP

JavaMOP is the state-of-the-art runtime verification framework for Java programs. Each JavaMOP specification contains a few components: event definition defined by using AspectJ-like language, formal properties which can be written in a few different formalisms, and error handler code to be executed when the underlying properties are violated. Each JavaMOP specification is compiled into AspectJ code, and at runtime the AspectJ code will be weaved into the original program. If during the programs' execution a violation is detected, the user defined handler code will be executed.

JavaMOP also has many advanced features:

- It is *parametric* as each event is not only bound to the method call, but also to the parameter objects and the receiver object. This gives JavaMOP the flexibility to define more complex properties.

- It is highly *optimized* by using the indexing trees and the weak reference map. As shown in our previous evaluation [63], JavaMOP only incurs relatively small performance overhead when being used to monitor hundreds of properties simultaneously.

- It supports various formalisms with different expressiveness, such as Extended Regular Expression (ERE), Context Free Grammar (CFG) and so on. Moreover, it also

supports user to define their own logic plugin. The underlying monitoring algorithms of JavaMOP does not depend on the specific formalism in use.

JavaMOP serves as the foundation for much of the work of this dissertation. In particular, EnforceMOP introduced in Chapter 3 is an extension of JavaMOP for enforcing properties at runtime for multithreaded programs.

2.2.2 Maximal Causal Model

Predictive analysis is an application of runtime verification techniques for multithreaded programs. The idea of predictive analysis is, given one execution trace of a multithreaded program, to build a model that represents the alternative executions (all the feasible executions) of the original program, and find errors such as data races and atomicity violations in those alternative executions. Even if the original execution does not trigger any error, predictive analysis may still find hidden errors in the alternative executions without the need to execute the program again. Because of its ability to find bugs, many researchers have used predictive analysis to find data races [24, 44], atomicity violations [83] and NullPointer Exceptions [32].

The core of predictive analysis is to build a model using the information gained from the current execution. The maximal causal model [82] contains the maximal number of executions that one can extract from one particular execution. It is proven to be maximal with two axioms: *prefix closedness* (events cannot be divided and should be generated in execution order) and *local determinism* (each event is determined by its prior events in the same thread). The details of the maximal causal model can be found in Chapter 4 and the proof of its maximality can be found in [82]. Because of its maximality, the maximal causal model outperforms other existing predictive models for detecting errors in multithreaded programs. In this dissertation, we are using the idea of the maximal causal model in another way for exploration of multithreaded programs: we systematically generate and explore all

the maximal causal models. Since each model represents a large (maximal possible) number of executions, in this dissertation it is shown that our technique provides great benefits for reducing the cost of state-space exploration of multithreaded programs.

Chapter 3

Enforcement of Testing Schedules and Runtime Properties

In this chapter we present our techniques for enforcing testing schedules and runtime properties for multithreaded programs. We first present the IMUnit framework for expressing and enforcing schedules for multithreaded unit tests; then we present the EnforceMOP framework for enforcing runtime properties. Our evaluation shows our techniques are able to enforce thread schedules and properties more effective and efficient compared to existing frameworks.

Note that the contributions presented in this chapter has already been published in the form of conference papers. IMUnit has been published at the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering 2011 (ESEC/FSE 2011) [46], EnforceMOP has been published at the International Symposium on Software Testing and Analysis 2013 (ISSTA 2013) [61]. The author of this dissertation is the main author or co-author of above papers, and would like to thank anonymous reviewers for valuable feedback.

3.1 Improved Multithreaded Unit Testing

This section presents the contributions of the IMUnit framework, which was developed to improve the expression and execution of multithreaded tests.

3.1.1 Example

We now illustrate IMUnit with the help of an example multithreaded unit test for the ArrayBlockingQueue class provided by the `java.util.concurrent` (JSR-166) package [49]. ArrayBlockingQueue is an array-backed implementation of a bounded blocking queue. One of the operations provided by ArrayBlockingQueue is `add`, which performs a non-blocking insertion of the given element at the tail of the queue. If `add` is performed on a full queue, it throws an exception. Another operation provided by ArrayBlockingQueue is `take`, which removes and returns the object at the head of the queue. If `take` is performed on an empty queue, it blocks until an element is inserted into the queue. These operations could have bugs that get triggered when the `take` and `add` operations execute on different threads. Consider specifying some scenarios for these operations (In fact, the JSR-166 TCK provides over 100 tests for various scenarios for similar classes).

```
1  @Test
2  public void testTakeWithAdd() {
3    ArrayBlockingQueue<Integer> q;
4    q = new ArrayBlockingQueue<Integer>(1);
5    Thread addThread = new Thread(
6      new CheckedRunnable() {
7        public void realRun() {
8          q.add(1);
9          Thread.sleep(150);
10         q.add(2);
11       }
12     }, "addThread");
13   addThread.start();
14   Thread.sleep(50);
15   Integer taken = q.take();
16   assertTrue(taken == 1 && q.isEmpty());
17   taken = q.take();
18   assertTrue(taken == 2 && q.isEmpty());
19   addThread.join();
20 }
```

(a) JUnit

Figure 3.1: Multithreaded unit test for ArrayBlockingQueue

Figure 3.1 shows a multithreaded unit test for ArrayBlockingQueue exercising the `add` and `take` operations in some scenarios. In particular, Figure 3.1(a) shows the test written as

a regular JUnit test method, with sleeps used to specify the required schedule. We invite the reader to consider what scenarios are specified with that test (without looking at the other figures). It should be clear that it is very difficult to understand which schedule is being exercised by reading the code of this unit test. While the sleeps provide hints as to which thread is waiting for another thread to perform operations, it is unclear which operations are intended to be performed by the other thread before the sleep finishes.

The test actually checks that `take` performs correctly both without and with blocking, when used in conjunction with `add` from another thread. To check both scenarios, the test exercises a particular schedule where the first `add` operation finishes before the first `take` operation starts, and the second `take` operation blocks before the second `add` operation starts. Line 14 shows the first sleep which is intended to pause the `main` thread[1] while the thread `addThread` finishes the first `add` operation. Line 9 shows the second sleep which is intended to pause the thread `addThread` while the `main` thread finishes the first `take` operation and then proceeds to block while performing the second `take` operation. If the specified schedule is not enforced during the execution of the test, it will result in false positives/negatives.

Figure 3.2(b) shows the same test logic written using MultithreadedTC [78]. Note that it departs greatly from the traditional JUnit where each test is a method. In MultithreadedTC, each test has to be written as a class, and each method in the test class contains the code executed by a thread in the test. The intended schedule is specified with respect to one global, logical clock. Since this clock measures time in *ticks*, we call the approach tick-based. When a thread executes a `waitForTick` operation, it is blocked until the global clock reaches the required tick value. The clock advances implicitly by one tick only when all the threads are blocked (and at least one blocked thread is executing a `waitForTick` operation). While a MultithreadedTC test does not rely on real time, and is thus more reliable than a sleep-based test, the intended schedule is still not immediately clear upon reading the test

[1] JVM names the thread that starts the execution `main` by default, although the name can be changed later.

```
1  @Test
2  @Schedule("afterAdd1->beforeTake1,
3              [beforeTake2]->beforeAdd2")
4  public void testTakeWithAdd() {
5    ArrayBlockingQueue<Integer> q;
6    q = new ArrayBlockingQueue<Integer>(1);
7    Thread addThread = new Thread(
8      new CheckedRunnable() {
9        public void realRun() {
10         q.add(1);
11         @Event("afterAdd1")
12         @Event("beforeAdd2")
13         q.add(2);
14       }
15     }, "addThread");
16   addThread.start();
17   @Event("beforeTake1")
18   Integer taken = q.take();
19   assertTrue(taken == 1 && q.isEmpty());
20   @Event("beforeTake2")
21   taken = q.take();
22   assertTrue(taken == 2 && q.isEmpty());
23   addThread.join();
24 }
```

(a) IMUnit

```
1  public class TestTakeWithAdd
2      extends MultithreadedTest {
3
4    ArrayBlockingQueue<Integer> q;
5
6    @Override
7    public void initialize() {
8      q = new ArrayBlockingQueue<Integer>(1);
9    }
10
11   public void addThread() {
12     q.add(1);
13     waitForTick(2);
14     q.add(2);
15   }
16
17   public void takeThread() {
18     waitForTick(1);
19     Integer taken = q.take();
20     assertTrue(taken == 1 && q.isEmpty());
21     taken = q.take();
22     assertTick(2);
23     assertTrue(taken == 2 && q.isEmpty());
24   }
25 }
```

(b) MultithreadedTC

Figure 3.2: ArrayBlockingQueue test written in IMUnit and MultithreadedTC

code. It is especially not clear when `waitForTick` operations are blocked/unblocked, because ticks are achieved implicitly when all the threads are blocked.

Figure 3.2(a) shows the same test logic written using IMUnit. The interesting events encountered during the test execution are marked with the `@Event` annotations, which our IMUnit tool properly translates into code for test execution. Since `@Schedule` annotations appear on methods, they are already fully supported in the current version of Java, and the intended schedule is specified with a `@Schedule` annotation that contains a comma-separated set of *orderings* among events. An ordering is specified using the binary operator `->`, where intuitively the left event is intended to execute before the right event. An event specified within square brackets denotes that the thread executing that event is intended to block after that event. It should be clear from reading the schedule that the thread `addThread` should finish the first `add` operation before the `main` thread starts the first `take` operation,

```
<Event Name>  ::= { <Id> "." } <Id>
<Thread Name> ::= <Id>
<Basic Event> ::= <Event Name> ["@" <Thread Name>]
                | "start" "@" <Thread Name>
                | "end" "@" <Thread Name>
<Block Event> ::= "[" <Basic Event> "]"
<Condition>   ::= <Basic Event>
                | <Block Event>
                | <Condition> "||" <Condition>
                | <Condition> "&&" <Condition>
                | "(" <Condition> ")"
<Ordering>    ::= <Condition> "->" <Basic Event>
<Schedule>    ::= { <Ordering> [","] }
```

Figure 3.3: Syntax of the IMUnit schedule language

and that the `main` thread should block while performing the second `take` operation before the thread `addThread` starts the second `add` operation.

We now revisit in the context of this example the issues with multithreaded unit tests listed in the introduction. In terms of *readability*, we believe that making the schedules explicit as in IMUnit allows easier understanding and maintenance of the schedules and code for both testing and debugging. In terms of *modularity*, note that IMUnit allows extracting the thread `addThread` as a helper thread (with its events) that can be reused in another test (In fact, many tests in the JSR-166 TCK [49] use such helper methods). Also, IMUnit allows specifying multiples schedules for the same test code. In contrast, reusing one of the thread methods from the MultithreadedTC test class becomes more involved, requiring subclassing tests, parametrizing tick values with variables, and providing appropriate values for those variables. In terms of *reliability*, IMUnit does not rely on real time and hence has no problems with unintended schedules. In terms of *migration costs*, note that the IMUnit test more closely resembles a traditional JUnit test than a MultithreadedTC test. This similarity eases the transition of legacy tests into IMUnit: in brief, add `@Event` annotations, add `@Schedule`, and remove `sleep` calls.

3.1.2 Schedule Language

We next describe the syntax and semantics of the language that is used in IMUnit's schedules.

Concrete Syntax

Figure 3.3 shows the concrete syntax of the implemented IMUnit schedule language. An IMUnit schedule is a comma-separated set of *orderings*. Each ordering defines a condition that must hold before a basic event can take place. A *basic event* is an event name possibly tagged with its issuing thread name when that is not understood from the context. An *event name* is any identifier, possibly prefixed with a package name (a list of dot-separated identifiers). There are two implicit event names for each thread, `start` and `end`, indicating when the thread starts and when it terminates. Any other event must be explicitly introduced by the user with the `@Event` annotation (see Figure 3.2(a)). A *condition* is a conjunctive/disjunctive combination of basic and block events, where block events are written as basic events in square brackets. A *block event* $[e']$ in the condition c of an ordering $c \to e$ states that e' must precede e and, additionally, the thread of e' is blocked when e takes place.

Schedule Logic

It is more convenient to define a richer logic than what is currently supported by our IMUnit implementation; the additional features are natural and thus may also be needed in future experiments. The semantics of our logic is given in Section 3.1.2; here is its syntax as a CFG:

$$
\begin{array}{rcl}
a & ::= & \mathit{start} \mid \mathit{end} \mid \mathit{block} \mid \mathit{unblock} \mid \text{event names} \\
t & ::= & \text{thread names} \\
e & ::= & a@t \\
\varphi & ::= & [t] \quad \mid \quad \varphi \to \varphi \quad \mid \quad \text{usual propositional connectives}
\end{array}
$$

The intuition for $[t]$ is "thread t is blocked" and for $\varphi \to \psi$ "if ψ held in the past, then φ must have held at some moment before ψ". We call these two temporal operators the

blockness and the *ordering* operators, respectively. For uniformity, all events are tagged with their thread. There are four implicit events. *start*@t and *end*@t were discussed above. The other two are *block*@t and *unblock*@t, corresponding to when t gets blocked and unblocked[2].

For example, the following formula in our logic

$$\begin{aligned} & (a_1@t_1 \wedge ([t_2] \vee (\neg(start(t_2) \rightarrow a_1@t_1)))) \rightarrow a_2@t_2 \\ \wedge \quad & (a_2@t_2 \wedge ([t_1] \vee (end(t_1) \rightarrow a_2@t_2))) \rightarrow a_2@t_2 \end{aligned}$$

says that if event a_2 is generated by thread t_2 then: (1) event a_1 must have been generated before that and, when a_1 was generated, t_2 was either blocked or not started yet; and (2) when a_2 is generated by t_2, t_1 is either blocked or terminated. As explained shortly, every event except for *block* and *unblock* is restricted to appear at most once in any execution trace. Above we assumed that $a_1, a_2 \notin \{block, unblock\}$.

Before we give the precise semantics, we explain how our current IMUnit language shown in Figure 3.3, whose design was driven exclusively by practical needs, fits as a fragment of the richer logic. An IMUnit schedule is a conjunction (we use comma instead of $\wedge$) of orderings, and schedules cannot be nested. Since generating *block* and *unblock* events is expensive, IMUnit currently disallows their explicit use in schedules. Moreover, to reduce their implicit use to a fast check of whether a thread is blocked or not, IMUnit also disallows the explicit use of $[t]$ formulas. Instead, it allows *block events* of the form $[a@t]$ (note the square brackets) in conditions. Since negations are not allowed in IMUnit and since we can show (after we discuss the semantics) that $(\varphi_1 \vee \varphi_2) \rightarrow \psi$ equals $(\varphi_1 \rightarrow \psi) \vee (\varphi_2 \rightarrow \psi)$, we can reduce any IMUnit schedule to a Boolean combination of orderings $\varphi \rightarrow e$, where φ is a conjunction of basic events or block events. All that is left to show is how block events are desugared. Consider an IMUnit schedule $(\varphi \wedge [a_1@t_1]) \rightarrow a_2@t_2$, saying that $a_1@t_1$ and φ must precede $a_2@t_2$ *and* t_1 is blocked when $a_2@t_2$ occurs. This can be expressed as

[2]It is expensive to explicitly generate *block*/*unblock* events in Java precisely when they take place, as it requires to poll each thread about its status; our currently implemented fragment only needs, through its restricted syntax, to check if a given thread is currently blocked or not, which is fast.

$((\varphi \wedge a_1@t_1) \rightarrow a_2@t_2) \wedge ((a_2@t_2 \wedge [t_1]) \rightarrow a_2@t_2)$, relying on the fact that $a_2@t_2$ can happen at most once.

Semantics

Our schedule logic is a carefully chosen fragment of *past-time linear temporal logic (PTLTL)* over special well-formed multithreaded system execution traces.

Program executions are abstracted as finite traces of events $\tau = e_1 e_2 \ldots e_n$. Unlike in conventional LTL, our traces are finite because unit tests always terminate. Traces must satisfy the obvious condition that events corresponding to thread t can only appear while the thread is alive, that is, between $start@t$ and $end@t$. Using PTLTL, this requirement states that for any trace τ and any event $a@t$ with $a \notin \{start, end\}$, the following holds

$$\tau \vDash \neg \diamondsuit (a@t \wedge (\diamondsuit end@t \vee \neg \diamondsuit start@t))$$

where $\diamondsuit$ stands for "eventually in the past". Moreover, except for $block@t$ and $unblock@t$ events, we assume that each event appears at most once in a trace. With PTLTL, this says that the following must hold ($\odot$ is "previously")

$$\tau \vDash \neg \diamondsuit (a@t \wedge \odot \diamondsuit a@t)$$

for any trace τ and any $a@t$ with $a \neq block$ and $a \neq unblock$.

The semantics of our logic is defined as follows:

$e_1 e_2 \ldots e_n \vDash e$ iff $e = e_n$

$\tau \vDash \varphi \ \wedge/\vee \ \psi$ iff $\tau \vDash \varphi$ and/or $\tau \vDash \psi$

$e_1 e_2 \ldots e_n \vDash [t]$ iff $(\exists 1 \leq i \leq n)$ $(e_1 = block@t$ and $(\forall i < j \leq n)$ $e_j \neq unblock@t)$

$e_1 e_2 \ldots e_n \vDash \varphi \rightarrow \psi$ iff $(\forall 1 \leq i \leq n)$ $e_1 e_2 \ldots e_i \nvDash \psi$ or

$(\exists 1 \leq i \leq n)$ $(e_1 e_2 \ldots e_i \vDash \psi$ and $(\exists 1 \leq j \leq i)$ $e_1 e_2 ... e_j \vDash \varphi)$

It is not hard to see that the two new operators $[t]$ and $\varphi \rightarrow \psi$ can be expressed in terms

of PTLTL as

$$[t] \equiv \neg \mathit{unblock@t}\ \mathcal{S}\ \mathit{block@t}$$

$$\varphi \rightarrow \psi \equiv \boxdot\neg\psi \vee \diamondsuit(\psi \wedge \diamondsuit\varphi)$$

where $\mathcal{S}$ stands for "since" and $\boxdot$ for "always in the past".

3.1.3 Enforcing & Checking

We now describe the IMUnit Runner, our tool for enforcing/checking schedules for IMUnit multithreaded unit tests. The tool executes each test for each IMUnit schedule (a test can have multiple schedules) and has two operation modes. In the *active mode*, it controls the thread scheduler to enforce an execution of the test that satisfies the given schedule. In the *passive mode*, it observes and checks the execution provided by the JVM against the given schedule.

Our runner is implemented using JavaMOP [23, 65], a high-performance runtime monitoring framework for Java. JavaMOP is generic in the property specification formalism and provides several such formalisms as *logic plugins*, including past-time linear temporal logic (PTLTL). Although our schedule language is a semantic fragment of PTLTL (see Section 3.1.2), enforcing PTLTL specifications in their full generality on multithreaded programs is a rather expensive problem.

Instead, we have developed a custom JavaMOP logic plugin for our current IMUnit schedule language from Figure 3.3. This plugin synthesizes a corresponding monitor that either enforces or checks a given IMUnit schedule, depending on the running mode. The monitor is infused within the test program by means of appropriate instrumentation in such a way that the schedule is enforced or checked at runtime, depending on the mode. Since JavaMOP takes care of all the low-level instrumentation and monitor integration details for us (after a straightforward mapping of IMUnit events into JavaMOP events), here we only briefly discuss our new JavaMOP logic plugin. It takes as input an IMUnit schedule and generates as output a monitor written in pseudo-code; a *Java shell* for this language then

```
1 switch (event){
2 case afterAdd1:
3   occurred_afterAdd1 = true; wakeAll();
4 case beforeTake2:
5   thread_beforeTake2 = currentThread();
6   occurred_beforeTake2 = true; wakeAll();
7 case beforeTake1:
8   while(!(occurred_afterAdd1))
9     wait();
10  occurred_beforeTake1 = true; wakeAll();
11 case beforeAdd2:
12  while(!(occurred_beforeTake2 && blocked(thread_beforeTake2)))
13    wait();
14  occurred_beforeAdd2 = true; wakeAll();
15 }
```

Figure 3.4: Monitor for the Schedule in Figure 3.2(a)

turns the monitor into AspectJ code, which is further woven into the test program. In the active mode, the resulting monitor enforces the schedule by blocking the violating thread until all the conditions from the schedule are satisfied. In the passive mode, it simply prints an error when its corresponding schedule is violated.

A generated monitor for an IMUnit schedule only observes the defined events. When an event e occurs, the monitor checks all the conditions that the event should satisfy according to the schedule, i.e., a Boolean combination of basic events and block events (Figure 3.3). The status of each basic event is maintained by a Boolean variable which is true iff the event occurred in the past. The status of a block event is checked as a conjunction of this variable and its thread's blocked state when e occurs. In the active mode, the thread of e will be blocked until this Boolean expression becomes true. If the condition contains any block event, periodic polling is used for checking thread states. Thus, IMUnit pauses threads only if their events are getting out of order for the schedule. This way, IMUnit allows both parallel execution and serialization, depending on the schedule. In the passive mode, the monitor will simply print an error message when any Boolean expression is false.

As an example, Figure 3.4 shows the active-mode monitor generated for the schedule in Figure 3.2(a). When events `afterAdd1` and `beforeTake2` occur, the monitor just sets the corresponding Boolean variables, as there is no condition for those events. For event

`beforeTake1`, it checks if there was an event `afterAdd1` in the past by checking the variable `occurred_beforeTake2`. The thread will be blocked until `afterAdd1` occurs. For event `beforeAdd2`, in addition to checking the Boolean variable for `beforeTake2`, it also checks whether the thread of the event `beforeTake2` is blocked. The thread of the event `beforeAdd2` will be blocked until both are satisfied.

3.2 EnforceMOP: A Runtime Property Enforcement System

In this section we present EnforceMOP, a novel framework for enforcing complex properties in multithreaded programs. The properties are enforced at runtime and do not require to modify the source code, so they can be modularly maintained. We show that EnforceMOP can be used effectively both in developing and in testing multithreaded programs.

3.2.1 Motivation

EnforceMOP can be used (1) to enforce general properties and (2) to enforce specific testing schedules in mutithreaded systems. Here we discuss two real world examples, one in each category, and show how EnforceMOP is used in each.

Enforcing General Properties

As stated in JavaDoc, an `ArrayList` is not allowed to be iterated and structurally modified at the same time [73].

> *The iterators returned by this class's iterator and listIterator methods are* fail-fast*: if the list is structurally modified at any time after the iterator is created, ...the iterator will throw a* `ConcurrentModificationException`*.*

However, it is very easy for developers to violate this. Moreover, it can be difficult to find and fix this error in multithreaded programs, because: (1) when using `ArrayList`, programmers are unaware of how it will be used in other threads; (2) the non-deterministic behavior of multithreaded programs makes it harder to reproduce and debug the problem. For example, as shown in a bug report in JFreeChart [71], one thread is iterating an `ArrayList` while another thread is attempting to call `add()` on the same `ArrayList` concurrently. As a result, a `ConcurrentModificationException` is non-deterministically thrown.

```
1  enforce SafeList_Iteration(Collection c, Iterator i) {
2      creation event create after(Collection c) returning(Iterator i) :
3          call(Iterator Iterable+.iterator()) && target(c) {}
4
5      event modify before(Collection c) :
6          (
7            call(* Collection+.add*(..)) ||
8            call(* Collection+.clear(..)) ||
9            call(* Collection+.offer*(..)) ||
10           call(* Collection+.pop(..)) ||
11           call(* Collection+.push(..)) ||
12           call(* Collection+.remove*(..)) ||
13           call(* Collection+.retain*(..))
14         ) && target(c) {}
15
16     event next before(Iterator i) :
17         call(* Iterator.next(..)) && target(i) {}
18
19     event hasnextfalse after(Iterator i) returning(boolean b) :
20         call(* Iterator+.hasNext()) && target(i) && condition(!b) {}
21
22     fsm :
23         na [
24             create -> init
25         ]
26         init [
27             next -> unsafe
28             hasnextfalse -> safe
29         ]
30         unsafe [
31             next -> unsafe
32             hasnextfalse -> safe
33         ]
34         safe [
35             modify -> safe
36             hasnextfalse -> safe
37             next -> safe
38         ]
39
40         @nonfail {}
41
42         @deadlock { System.out.println("Deadlock detected!"); }
43 }
```

Figure 3.5: Safe List Iteration Specification

We can easily state the property of safe iteration in JavaMOP [22, 23], as shown in Figure 3.5 (ignore the gray areas for now, which are parts of the EnforceMOP extension). Monitoring-oriented programming (MOP) is a generic multi-formalism monitoring framework, which takes an implementation and a set of specifications as input, and checks whether the implementation violates the specifications at run time. JavaMOP is the Java instance

of MOP, currently using AspectJ [54] for event specification and instrumentation. As shown in Figure 3.5, a JavaMOP specification consists of four parts. The first is the specification header, with modifiers and parameters. Each parameters instance yields a monitor instance. Here, the `Collection` and `Iterator` parameters indicate that a different monitor will be generated for each combination of instances of these two parameters. Monitors corresponding to different parameter instances will not interfere with each other. More details can be found in [22, 23]. The second part describes all the relevant events, which serve as an abstraction of the running program. Those events drive the monitor from one state to another state.

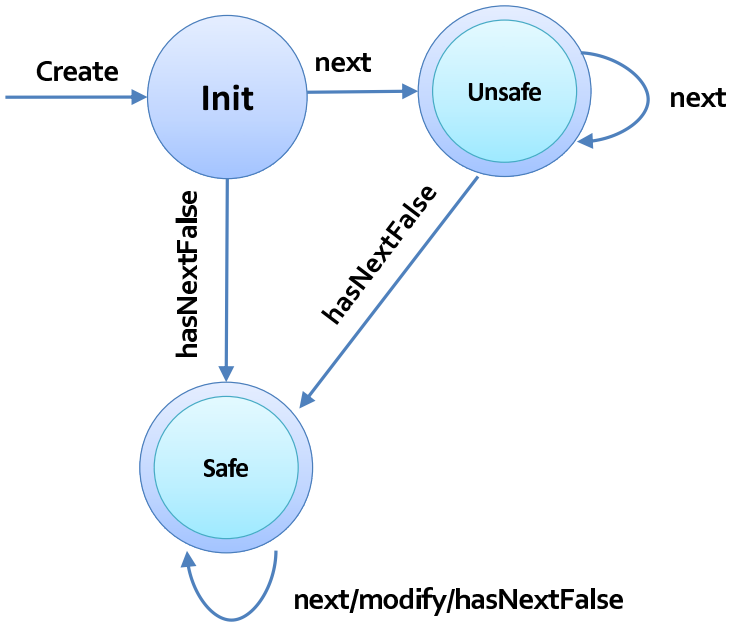


Figure 3.6: Safe List Iteration FSM

The third part is the actual property, starting with the *logic plugin* in which it is stated. In Figure 3.5 we use the finite state machine (FSM) plugin to state the property depicted in Figure 3.6. A monitor begins with the `Init` state after an iterator is created for an `ArrayList` instance. Now if `next()` is called on the iterator then the monitor enters the `Unsafe` state. Any transitions not defined in the FSM will cause the monitor to enter a default `fail` state,

indicating `ArrayList` was modified while an iteration is in progress. Method `hasNext()` returns false when the iterator has finished its job (we assume `hasNext()` is always called before `next()`, which is common practice), generating event `hasNextFalse` that makes the monitor enter its `Safe` state, indicating that modifications to the `ArrayList` are now allowed.

EnforceMOP has been purposely designed to require minimal learning effort from existing JavaMOP users. It should take less than one minute to change an existing JavaMOP specification into an EnforceMOP specification that enforces rather than monitors the former in multi-threaded systems. First, one needs to use the new `enforce` modifier (grayed in Figure 3.5). Second, one has to specify the desired state or group of states which the monitor should *not* be allowed to leave. Third, one may optionally use the new `@deadlock` handler to provide code to be executed in case of deadlock. We discuss the latter two in more detail below.

EnforceMOP enforces monitors to remain in certain states by controlling thread schedules. JavaMOP already allows users to associate code to monitor states, to be executed when the monitor reaches those states. Using the same notation, EnforceMOP enforces the monitor to never leave the specified states. Each logic plugin provides and documents its own monitor state names. The FSM plugin allows users to define and name groups of states, and provides a predefined group of states named `nonfail` including all the states except `fail`. In our example, we state that we want EnforceMOP to never allow the monitors to leave their `nonfail` group of states. If a monitor attempts to execute a transition not shown in Figure 3.6, for example execute event `modify` in state `unsafe`, the thread scheduling code generated by EnforceMOP will block the unsafe thread and thus guarantee safe iteration behaviors. For example, when one thread is iterating over the list so the monitor is in the `unsafe` state, any other thread attempting to modify the same list will get blocked until the end of the iteration is reached; then they are unblocked and allowed to perform their modifications.

Since EnforceMOP blocks threads during execution, it may directly or indirectly cause

```
1  @Test
2  public void testPutWithTake() throws InterruptedException {
3      final SynchronousQueue q = new SynchronousQueue();
4      Thread t = new Thread(new CheckedRunnable() {
5              public void realRun() throws InterruptedException {
6                  int added = 0;
7                  try {
8                      while (true) {
9                          q.put(added);
10                         ++added;
11                     }
12                 } catch (InterruptedException success) {
13                     assertEquals("PutWithTake", 1, added);
14                 }
15             }}, "putThread");
16     t.start();
17     Thread.sleep(SHORT_DELAY_MS);
18     assertEquals("PutWithTake",0, q.take());
19     Thread.sleep(SHORT_DELAY_MS);
20     t.interrupt();
21     t.join();
22 }
```

Figure 3.7: Original SynchronousQueue Test in TCK

program deadlock. For example, when the specified property is impossible to enforce (that is, any thread schedule yields an execution that violates the property), all threads will eventually block, resulting in a deadlock. The `@deadlock` handler tells the monitor what to do when a deadlock occurs. Here we chose to output an error message when a deadlock happens, but in general one can execute any code (shutdown the system, restart a certain thread, etc).

Enforcing Specific Testing Schedules

When writing a unit test for a multithreaded program, it is vital to have the ability to specify and enforce a desired thread schedule when running that test. Consider the real-life multithreaded test in Figure 3.7, borrowed from the TCK unit tests of `SynchronousQueue` in `java.util.concurrent`. `SynchronousQueue` is a special kind of queue where the thread executing `put` blocks when the queue is full and the thread executing `take` blocks when the

queue is empty. Thread `putThread` is calling `put` inside a loop to fill the queue. When the queue is full, `putThread` blocks. The desired thread schedule is: the main thread first waits for `putThread` to block, then takes one element and checks it (line 18), then waits for `putThread` to block again, and then interrupts it. This schedule is achieved in the TCK unit test using `sleep` statements, which as discussed in [46] and in Section 3.2.3 are non-modular, unreliable and slow.

EnforceMOP is an ideal vehicle to enforce specific testing schedules for multithreaded unit tests. The idea is to *separate* the functionality of the unit test from the desired schedule, and to implement the former as an unrestricted program (e.g., by removing the grayed sleep statements in Figure 3.7) and to enforce the latter with EnforceMOP. Figure 3.8 shows the EnforceMOP specification of the schedule meant in Figure 3.7. The event `beforeput` is generated right before calling method `put`, and events `beforeinterrupt` and `beforetake` right before calling methods `interrupt` and `take`, respectively. EnforceMOP defines a new pointcut, `threadBlocked`, telling the thread that is executing the event to *wait until* the specified thread is blocked. In this example, when the main thread is about to call the method `take` or `interrupt`, it waits until `putThread` gets blocked. We used the Extended Regular Expression (ERE) plugin (+ means one or more repetitions) to specify the actual schedule (line 19). Thus, the main thread blocks before it calls the method `take` until event `beforeput` occurs at least once and `putThread` blocks, then it unblocks and checks the assertion, and then it blocks again before it calls the `interrupt` until `beforeput` occurs and `putThread` blocks. The desired schedule is thus specified modularly, reliably and, as seen in Section 3.2.3, efficiently.

As discussed later in this chapter, it is not easy to use existing multithreaded testing frameworks to specify this particular schedule, because it involves a loop. EnforceMOP can support repeating events in a thread schedule using the bare capabilities of the its logic plugins, e.g., the ERE +.

EnforceMOP has been implemented independent of specification formalisms to support

```
1  enforce SynchronousQueueTest_testPutWithTake() {
2
3    String putThread = "";
4
5    event beforeinterrupt before() :
6      call(* Thread+.interrupt()) && threadBlocked(putThread){}
7
8    event beforetake before() :
9      call(* SynchronousQueue+.take()) && threadBlocked(putThread){}
10
11   event beforeput before() :
12     call(* SynchronousQueue+.put(..)) {
13             if (putThread.equals("")) {
14                 putThread = Thread.currentThread().getName();
15             }
16     }
17
18
19   ere : beforeput+ beforetake beforeput+ beforeinterrupt
20
21   @nonfail {}
22
23   @deadlock {System.out.println("Deadlock detected!");}
24 }
```

Figure 3.8: EnforceMOP Schedule for Test in Figure 3.7

enforcing arbitrarily complex properties. Those properties can be application-independent (such as the safe list iteration property above) or application-specific (such as the specific schedule in the multithreaded unit test above). Different property specification formalisms have different expressiveness, and the flexibility to use any of them helps users specify a wide variety of properties precisely and elegantly. For example, we show that FSM cannot express some useful properties which are expressible with other formalisms. Additionally, EnforceMOP supports parametric specifications, so different (enforcing) monitor instances are created for different parameter instances.

EnforceMOP can be thought of as a *semantic*-based synchronization approach, complementary to the traditional *syntax*-based synchronization approach: the semantics is embodied in the formal specification for each property. EnforceMOP allows developers to declaratively and modularly state the actual properties they want to enforce in their pro-

grams, and thus by avoiding over-synchronization it has the potential to be more efficient than traditional synchronization mechanisms, as empirically shown later in this chapter.

3.2.2 Approach and Implementation

Here we give an overview of EnforceMOP, with particular emphasis on how it smoothly integrates with JavaMOP. The key challenge of this integration was to design the enforcement mechanisms in a formalism-independent way. Figure 3.9 recalls the overall architecture of JavaMOP. It consists of a Java-specific client and language-independent logic plugins. The logic plugin manager makes available to the client various logic plugins (discussed shortly), by taking as input a formula written in a specific logic and outputting language-independent monitoring pseudocode. This pseudocode is then used to generate Java and AspectJ code, which is finally woven into the original program to monitor.

Logic Plugins and Enforcement Categories

Each EnforceMOP specification requires a property over the specified events, formalized using one of the available logic plugins. Some formalisms are more convenient or more efficient than others in some situations. EnforceMOP currently supports for enforcement all the logic formalisms supported by JavaMOP for monitoring. We briefly recall them:

Finite State Machine (FSM): A finite state machine consisting of a set of states and a set of state transitions. Each transition is triggered by an event.

Extended Regular Expression (ERE): A regular expression extended with complement; each letter is an event.

Linear Temporal Logic (LTL): A future time linear temporal logic formula describing good or bad prefixes.

Past Time Linear Temporal Logic (PTLTL): A linear temporal logic formula with temporal operators referring to the past states of the execution trace.

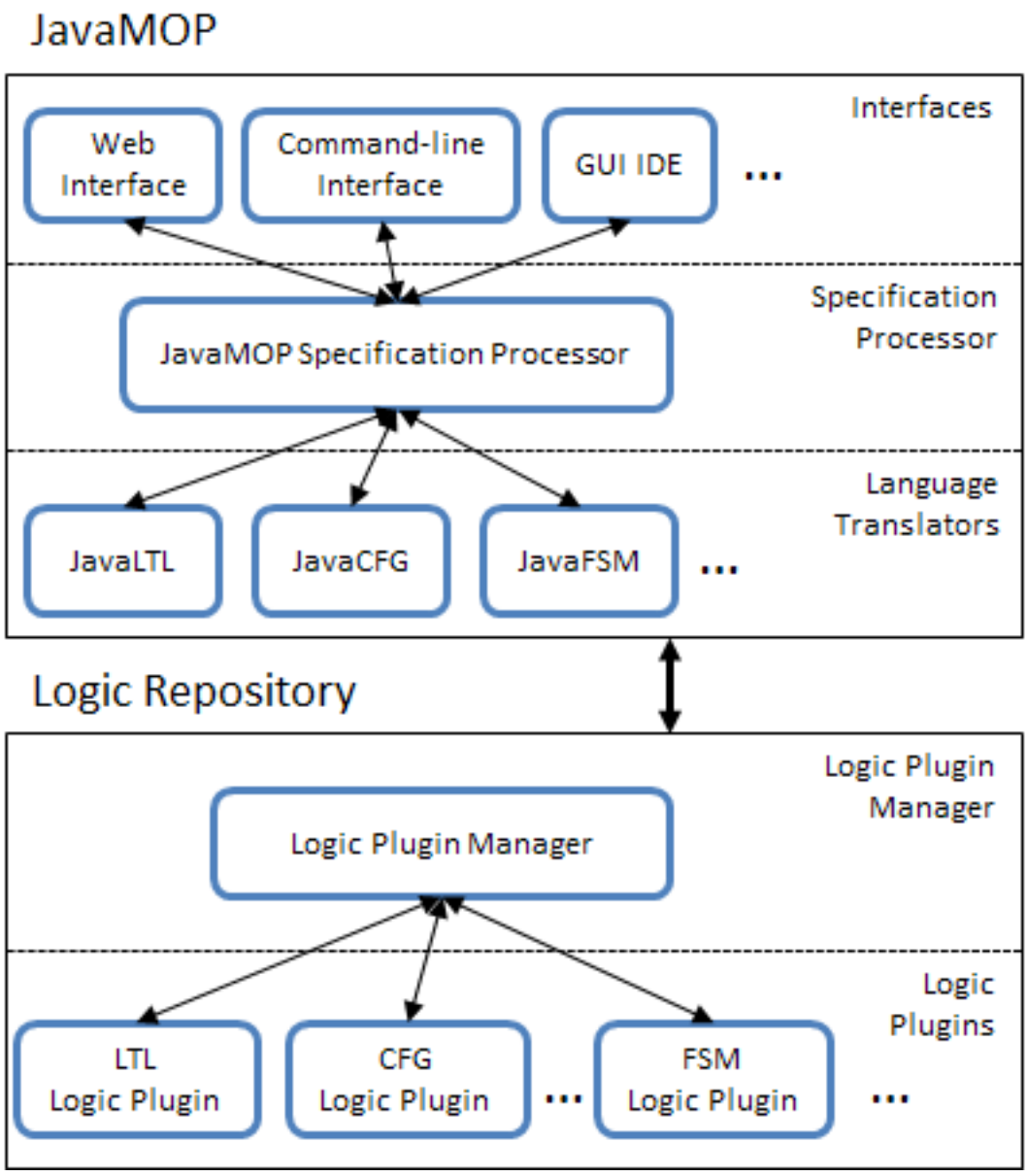


Figure 3.9: JavaMOP Overall Architecture

Context Free Grammar (CFG): A context free grammar defined in BNF, where each terminal is an event.

String Rewriting System (SRS): Turing-complete string rewriting formalism, where each alphabet is an event.

Once a specific logic formalism is chosen, the next step is to choose in which way the property is enforced. For example, one can specify the correct behaviors of the system, and enforce the monitor to always obey the specification; alternatively, one can specify the adverse behaviors of a system, and enforce the monitor to never satisfy the specification. To accommodate all the existing logic plugins, EnforceMOP provides a set of pre-defined categories (a category can be viewed as a set of monitor states) to be enforced. As shown in Table 3.1, different logic formalisms have different corresponding categories. We describe each pre-defined category:

***fail*:** When the monitor encounters an event not accepted in the current state (in FSM), or the current trace does not match any prefix of the given pattern (in ERE and CFG). In SRS, *fail* can be defined by the user.

***nonfail*:** The opposite of *fail*, when the incoming event is accepted by the current state, or the current trace matches one prefix of the given pattern.

***succeed*:** In SRS, *succeed* is defined by the user to trigger when certain patterns are matched.

***match*:** Corresponds to a situation wherein the trace matches the entire specified pattern.

***nonmatch*:** Corresponds to a situation wherein the trace does not match the entire specified pattern.

***violation*:** Occurs when the trace is not a prefix of any trace that satisfies the given formula in LTL and PTLTL.

***validation*:** Corresponds to a situation wherein the trace satisfies the given formula in PTLTL.

Some plugins allow users to define their own categories, which can then be enforced using EnforceMOP. For example, FSM allows to define an alias of a group of states. EnforceMOP can then enforce the monitor to stay in one of those.

Logic	Support Categories
FSM	*fail*/*nonfail*
ERE	*fail*/*nonfail*/*match*/*nonmatch*
LTL	*violation*
PTLTL	*violation*/*validation*
CFG	*fail*/*nonfail*/*match*/*nonmatch*
SRS	*fail*/*nonfail*/*succeed*

Table 3.1: Predefined Categories for each Logic Plugin

The Property Enforcing Algorithm

The key challenge in the design and development of EnforceMOP was to engineer its enforcement mechanism to work in a logic-formalism-independent way, to allow its users to choose any of the specification formalisms above for their properties and to enforce any of their categories. The problem is that different logic formalisms have different underlying representations of their monitors; for example, FSM uses lists of arrays to represent states and transitions, while CFG uses stacks to represent push down automata. However, all monitors share a common interface: take any given event and trigger a corresponding (logic-specific) transition.

The key idea of our monitor-independent enforcing algorithm is quite simple: use the common interface with a *clone* of the original monitor to decide whether to allow the current event to be executed on the original monitor, or to block the current thread. The algorithm is presented in Figure 3.10. The new event is sent to the cloned monitor, to check using its logic-specific semantics, which is irrelevant to EnforceMOP, whether the property we want to enforce would be violated if we let the event go through. If yes, then we block the current thread. If not, then it is safe for the original monitor to execute this event, so we let the event go through. We invoke the blocked thread and repeat the process above whenever a new event is generated in any other thread. Since a monitor is shared between different threads, its status may be changed by events executed in other threads. Whenever we find out that executing the pending event on the cloned monitor will not violate the property we want to enforce, we will unblock the thread and resume its execution.

```
1  // Inputs
2  Set<Category> violationCategories;
3  Event event;
4  Monitor origMonitor;
5
6  void enforceProperty() {
7        do {
8              clonedMonitor = origMonitor.clone();
9              clonedMonitor.execute(event);
10             if (clonedMonitor.status ∈ violationCategories) {
11                 clonedMonitor = null; // for garbage collection
12                 wait;
13             }
14             else {
15                 clonedMonitor = null; // for garbage collection
16                 break;
17             }
18       } while (true);
19       origMonitor.execute(event);
20       notify all waiting monitors;
21 }
```

Figure 3.10: Algorithm for Enforcing Properties

Deadlock Detection

When enforcing a property, it could be possible that all the threads get blocked by EnforceMOP, so the program deadlocks. This happens when the program reaches a state in which any event to be executed by any thread would violate the property. Since property violations can mean anything depending upon the application and the property, our approach is to provide the mechanism and let the user decide how to use it, that is, how to proceed at deadlock. Specifically, EnforceMOP provides an on-the-fly deadlock detection mechanism which works as follows. Every newly started thread is recorded in a global map. A separate deadlock detection thread checks this map periodically. When all the threads in the map are blocked, a deadlock occurred. The *@deadlock* handler serves like any other JavaMOP handlers, so users can take arbitrary actions when a deadlock happens; for example, restart the system or print error messages.

Implementation

We implemented EnforceMOP in Java as an extension of JavaMOP. JavaMOP takes a property file as input and generates an AspectJ file that contains monitor, recovery and instrumentation code, which is then compiled and woven into the original program using any AspectJ compiler. We added *enforce* as a new keyword modifier to JavaMOP properties, in a way that any existing JavaMOP property can be turned into an EnforceMOP by only adding the *enforce* modifier. To generate code to enforce a property, we extended the code generator in JavaMOP with a new class, `EnforceMonitor`, which is responsible for generating all the code to enforce a property when the *enforce* modifier is used.

As already noted in previous work on specifying thread schedules [46, 78], it is crucial to have the ability to trigger an event when a specific thread gets blocked. For that reason, we added a new pointcut to EnforceMOP, `threadBlocked`. It takes a thread name as argument and triggers an event in the monitor only when that specific thread is blocked. We implemented this by using the `threadStart` pointcut of JavaMOP to add any thread to a global thread map when it starts. Then `threadBlocked` is easily implemented by polling the state of that specific thread in the map.

3.2.3 Applications and Evaluation

We envision EnforceMOP to be used: (1) as a framework to enforce general complex safety properties at runtime; and (2) as a testing framework to enforce specific schedules when unit testing multithreaded applications. We next evaluate the effectiveness of EnforceMOP in these two aspects. We first present a number of applications using EnforceMOP to enforce general properties, then we use it to enforce specific testing thread schedules and compare it with several other multithreaded testing frameworks.

Enforcing General Properties

Safe Iteration As shown in Figures 3.5 and 3.6, EnforceMOP can be used to guarantee safe iteration of a collection in multithreaded programs. Motivated by a real bug in JFreeChart [71], we used EnforceMOP to specify and enforce correct behaviors of iterating a collection in multithreaded programs. In the test case attached with the bug report, two threads are created and one of them adds a new element to the collection while the other iterates through the collection. These two actions are repeated many times, so in the original program the `ConcurrentModificationException` is thrown almost every time when the test case executes. After we applied the property in Figure 3.5 using EnforceMOP, the exception never gets thrown after 100 times of execution of the same test case.

Mutual Exclusion

Another bug in JFreeChart [72] is caused by concurrent execution between any modification method and `hashCode` on the same `ArrayList`. The root cause of this bug is similar to the previous one: JDK's `hashCode` method iterates through all the elements of the list in order to compute the hash value of the whole list. So a `ConcurrentModificationException` will be thrown if `hashCode` and any other modification method are called at the same time. However, since the iteration of the list is encapsulated in `hashCode`, what users actually want is the mutual exclusion *only* between the execution of `hashCode` and the execution of other modification methods. This cannot be easily done using Java synchronization mechanisms. Suppose we only want the execution of `hashCode` and any other modification method to be mutually exclusive, but any other pairs of methods to be allowed to execute concurrently. If we blindly use the `synchronized` keyword on all these then all the methods become mutually exclusive of each other, thus over-synchronizing the program and harming performance (for example, two threads could safely execute `hashCode` concurrently).

One can try to use a `ReadWriteLock` from `j.u.c` instead, for example to use `ReadLock` in `hashCode` and `WriteLock` in all modification methods. However, concurrent execution

```
1  enforce SafeListCFG(List l) {
2
3      event beforehashcode before(List l) :
4          call(* Object+.hashCode(..)) && target(l) {}
5
6      event afterhashcode after(List l) :
7          call(* Object+.hashCode(..)) && target(l) {}
8
9      event beforemodify before(List l) :
10         (
11                 call(* List+.add*(..)) ||
12                 call(* List+.remove(..)) ||
13                 call(* List+.retain*(..)) ||
14                 call(* List+.clear(..)) ||
15                 call(* List+.set*(..))
16         ) && target(l) {}
17
18     event aftermodify after(List l) :
19         (
20                 call(* List+.add*(..)) ||
21                 call(* List+.remove(..)) ||
22                 call(* List+.retain*(..)) ||
23                 call(* List+.clear(..)) ||
24                 call(* List+.set*(..))
25         ) && target(l) {}
26
27     cfg :
28         S -> A S | B S | epsilon,
29         A -> A beforehashcode A afterhashcode | epsilon,
30         B -> B beforemodify B aftermodify | epsilon
31
32     @nonfail {}
33
34     @deadlock { System.out.println("Deadlock detected!"); }
35 }
```

Figure 3.11: Mutual Exclusion between HashCode and List Modification using CFG

between any two modification methods would still be prohibited, thus reducing the potential for parallelism[3]. Mutual exclusion is a common property desired in multithreaded programs, but without careful consideration it is very easy to over-synchronize and thus hurt the performance.

[3]The concurrent use of `ArrayList` is known to be problematic; one should instead use concurrent data-structures from `j.u.c`. We use it here only to show how to enforce mutual exclusion between groups of methods with EnforceMOP.

Figure 3.11 shows how to enforce mutual exclusion for this specific case using EnforceMOP with the CFG plugin. The property is parametric in the list, so operations on different list instances will not interfere with each other. Since we want to enforce mutual exclusion between method calls, we use both `before` and `after` pointcuts to describe events. There are four types of events in this property: `beforehashcode` and `afterhashcode` indicate the start and end of the execution of `hashCode`, and `beforemodify` and `aftermodify` represent the start and end of all the modification methods on `ArrayList`. The property is defined using a CFG, which allows us to pair the start and the end events of the execution of `hashCode` or of modification methods. While the execution of `hashCode` is in progress (event `afterhashcode` has not been encountered), the execution of any modification methods is not allowed (event `beforemodify` is not allowed).

Although the SRS plugin is the most expressive formalism available with EnforceMOP (it is Turing-complete), we often found it in our experiments that SRS is quite convenient to specify even simpler properties. For example, we can replace the CFG in Figure 3.11 with the following equivalent SRS:

```
srs :
   beforemodify aftermodify -> #epsilon .
   beforehashcode afterhashcode -> #epsilon .
   beforemodify afterhashcode -> #fail .
   beforehashcode aftermodify -> #fail .
   beforemodify beforehashcode -> #fail .
   beforehashcode beforemodify -> #fail .
```

The SRS rules apply on the trace as it is being generated to keep it in a canonical form. In our case, consecutive event pairs `beforehashcode` and `afterhashcode`, and `beforemodify` and `aftermodify`, will dissolve (`#epsilon` is the empty string), and the other four event pairs will force the monitor to fail. In Figure 3.11 we enforce the monitor to never enter

its *fail* state (line 32), so whenever a thread wants to call a modification method while a `hashCode` method call is in progress, EnforceMOP will block that thread. Note that we only make `hashCode` and the group of the modification methods mutually exclusive, but no more than that. For example, the sequences `beforehashcode beforehashcode afterhashcode afterhashcode` and `beforemodify beforemodify aftermodify aftermodify` are both accepted. This allows maximum parallelism in the program. Note that this property cannot be expressed with FSM because the numbers of method start and end events should match, and FSM does not have the expressiveness to count the number of occurrences of events. But it can be elegantly specified with CFG or SRS, showing the advantage of supporting multiple logic formalisms.

Read Write Lock

Here we address a performance problem in Log4J [9] caused by over-synchronization. The class `Category` is supposed to be thread safe, so the `synchronized` keyword is used in many of its methods (`append`, `addAppender` and `removeAppender`). Each `Category` object has a list of `appenders`; the method `append` calls methods on all the elements in the list but it does not modify the list itself. The `synchronized` keyword guarantees the mutual exclusion between any methods, but it is not needed when two threads are both executing `append`. In the bug report [9] one developer mentioned *"...observing plenty of threads waiting on this synchronization..."*.

We completely removed the usage of `synchronized` and used EnforceMOP instead to specify precisely the desired synchronization between those method pairs. The property is written using SRS and is shown in Figure 3.12. We first group the methods into two sets: methods that will not modify the list (`append`) and methods that will modify the list (`addAppender`, `removeAppender` and `removeAllAppenders`). Then we define events to mark the start and end of those methods. The property is similar to the previous one, except one more rule: `beforemodify beforemodify -> #fail`. This prevents two modification methods (e.g., `addAppender` and `removeAppender`) from happening in parallel, to avoid

```
1  enforce SafeAppendSRS(Category c) {
2
3      event beforeappend before(Category c) :
4          call(* Category+.append(..)) && target(c) {}
5
6      event afterappend after(Category c) :
7          call(* Category+.append(..)) && target(c) {}
8
9      event beforemodify before(Category c) :
10         (
11         call(* Category+.addAppender(..)) ||
12         call(* Category+.removeAppender(..)) ||
13         call(* Category+.removeAllAppenders(..))
14         ) && target(c) {}
15
16     event aftermodify after(Category c) :
17         (
18         call(* Category+.addAppender(..)) ||
19         call(* Category+.removeAppender(..)) ||
20         call(* Category+.removeAllAppenders(..))
21         ) && target(c) {}
22
23     srs :
24         beforemodify aftermodify -> #epsilon .
25         beforeappend afterappend -> #epsilon .
26         beforemodify afterappend -> #fail .
27         beforeappend aftermodify -> #fail .
28         beforemodify beforeappend -> #fail .
29         beforeappend beforemodify -> #fail .
30         beforemodify beforemodify -> #fail .
31
32     @nonfail {}
33
34     @deadlock { System.out.println("Deadlock detected!"); }
35 }
```

Figure 3.12: Mutual Exclusion Property between Method Pairs in Log4J using SRS

inconsistency. We disallow the parallel execution of any modification method and `append`, but we do allow parallel execution between `append` methods. This increases parallelism and matches the intention of the developer.

We wrote a test case to reproduce the performance problem in [9] caused by over-synchronization. We created 50 threads, half of them calling method `append` and another half calling methods `addAppender` and `removeAppender` in parallel. We collected running

No Sync	Original (Over-Sync)	EnforceMOP	`ReadWriteLock`
44.4	500.8	49.7	221.3

Table 3.2: Test execution time (ms) for different synchronization mechanisms

time with the following configurations: the original over-synchronized code with the use of `synchronized`; EnforceMOP enforcing proper synchronization as shown in Figure 3.12; a `ReadWriteLock` implementation proposed by the developer in the bug report and the original code with all `synchronized` keywords completely removed as a base line to show the performance overhead. For each configuration we run the test case 10 times and get the average running time. Results are shown in Table 3.2.

From the results we can see that EnforceMOP performs much better than the original over-synchronized version, since it increases the maximal parallelism of the application. EnforceMOP also outperforms `ReadWriteLock`. Since the parallelism allowed by EnforceMOP is the same as with `ReadWriteLock` , we think the reason for our better performance is due to the fact that `ReadWriteLock` in Java involves calling a lot of library code and maintaining the lock status (since it's reentrant), while JavaMOP is highly optimized.

Dining Philosophers

Five philosophers sit next to each other around a round table. There are five forks placed between each pair of adjacent philosophers. Each philosopher needs to pick up the two forks around him to eat and they are allowed to eat at the same time. Each fork can only be used by one philosopher at any time. A deadlock happens when each philosopher picks up a different fork at the same time, and all of them are attempting to pick the other to start eating.

To implement the dining philosophers problem, locks are typically used to enforce the property that one fork can only be taken by one philosopher at any time. Instead, we first use EnforceMOP to enforce this property, so no synchronization code is needed in the program at all. Then we enforce the property stating that at most four philosophers can eat at the

```
1 public class Phil implements Runnable {
2
3     public Fork leftFork, rightFork;
4
5     public void getLeftFork() { leftFork.acquire(); }
6     public void releaseLeftFork() { leftFork.release(); }
7     ...
8
9     public void run() {
10        getLeftFork();
11        getRightFork();
12        eat();
13        releaseLeftFork();
14        releaseRightFork();
15    }
16 }
```

Figure 3.13: Source code of dining philosophers without synchronization

same time, which guarantee deadlock freedom.

Synchronization Free Implementation

The sketch of our code is shown in Figure 3.13. Each philosopher is represented by a `Runnable` object and runs concurrently. A philosopher stars eating after he grabs his left fork first and then his right fork next. Then he also releases his left fork first and his right fork second. There is no synchronization or lock used in the source code, so the correctness property of dining philosophers—no folk can be taken by two philosophers at the same time—is not guaranteed.

We use EnforceMOP to enforce the property of exclusive use of forks shown in Figure 3.14. This property is parametrized by a `Fork` instance. Event `acquire` corresponds to the start of method call `acquire` on a `Fork` instance and event `release` to the end of method call `release`. This property guarantees that when a fork is being used, it cannot be used again until released. Any other thread attempting to call `acquire` on a `Fork` instance at state `busy` will be blocked. Thus EnforceMOP yields a correct and elegant implementation of the dining philosophers problem without any explicit synchronization mechanism.

Deadlock Free Property

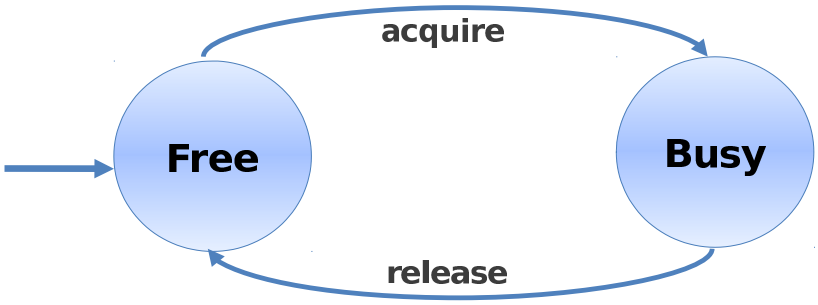


Figure 3.14: Exclusive use of Forks Property in FSM

The above property only guarantees the correct usage of forks. Deadlock is possible when each philosopher takes his left fork at the same time. We use EnforceMOP to enforce a property to avoid deadlock in our implementation, as shown in Figure 3.15. The idea behind it is that we only allow at most four philosophers to attempt to eat at the same time, so at least one philosopher would be able to grab both forks and eat. Then he will release both forks and other philosophers will be able to eat. Event `useLeftFork` marks the start of method call `getLeftFork`, and event `releaseLeftFork` marks the end of method call `releaseRightFork`. Each state serves as a *counter* of how many philosophers are attempting to eat. At state `Four` any philosopher attempting to grab forks will be blocked until a previous philosopherreleases all his forks. Unlike the previous property in Figure 3.14, this deadlock avoidance property is *not* parametric. It serves as a central coordinator to coordinate philosophers. With this property and the previous property in Figure 3.14, we are able to enforce the correct behavior of dining philosophers and avoid deadlocks at the same time, without using any synchronization in the source code.

Fair Thread Scheduler

In multithreaded programs *fairness* is a property of a thread scheduler which ensures every thread gets its turn to execute eventually. In practice the lack of fairness may cause thread starvation bugs [12, 45].

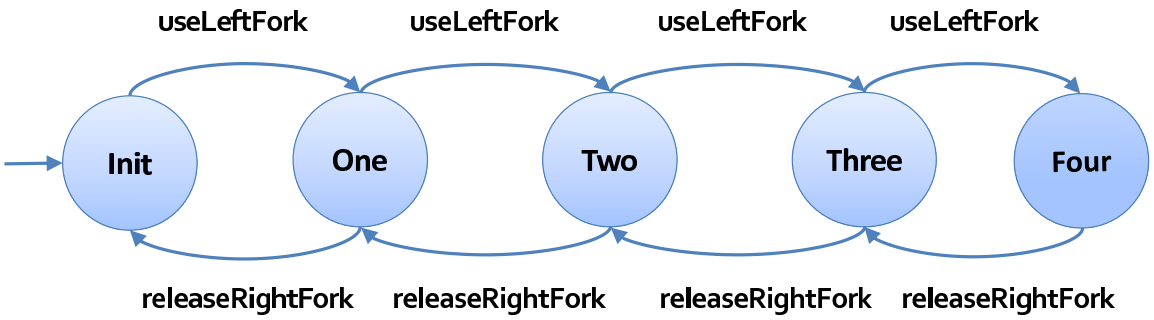


Figure 3.15: Deadlock Avoidance Property in FSM

```
1  enforce FairScheduler(Task t) {
2
3      event workone before(Task t) :
4          call(* Task+.doWork(..)) && threadName("t1") && target(t) {}
5
6      event worktwo before(Task t) :
7          call(* Task+.doWork(..)) && threadName("t2") && target(t) {}
8
9      event afterwork after(Task t) :
10         call(* Task+.doWork(..)) && target(t) {}
11
12     fsm :
13         Init [ workone -> ExecOne
14                worktwo -> ExecTwo ]
15         ExecOne [ afterwork -> OneDone ]
16         ExecTwo [ afterwork -> TwoDone ]
17         OneDone [ worktwo -> Finish ]
18         TwoDone [ workone -> Finish ]
19         Finish [ afterwork -> Init ]
20
21     @nonfail {}
22
23     @deadlock { System.out.println("Deadlock detected!"); }
24 }
```

Figure 3.16: Fair Scheduler Property

Here we show how EnforceMOP can be used to ensure a simple thread scheduling fairness property. Consider a program executing two threads, where each thread executes a loop with the same number of iterations. In each loop iteration a method `doWork` is called. Inside the `doWork` method each thread sleeps a random interval of time (this is meant to simulate real scenarios where workload is unknown). If we run the program without controlling the

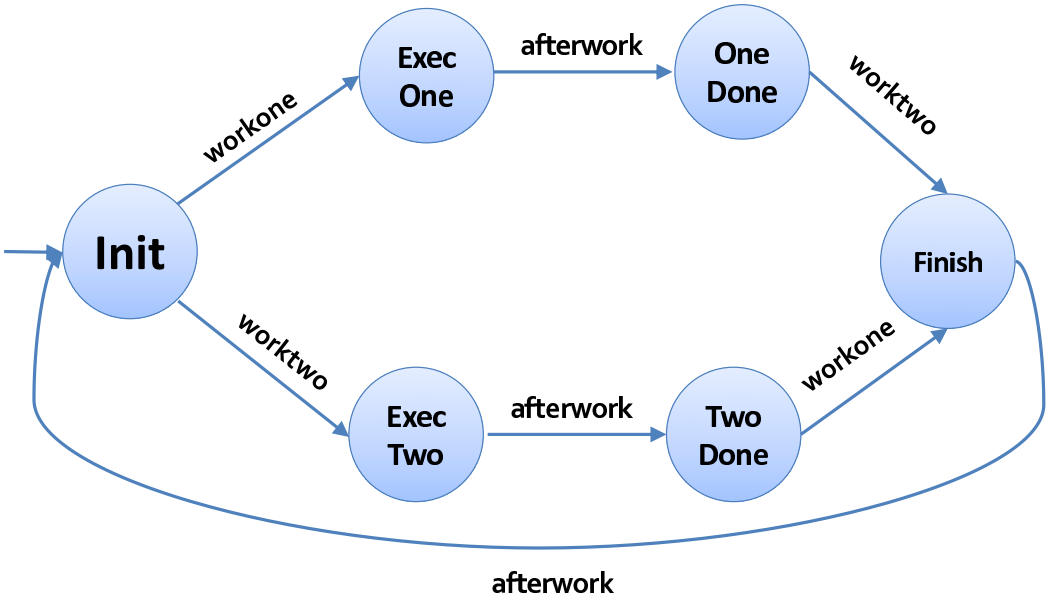


Figure 3.17: Fair Scheduler FSM

schedules, it is possible that one thread progresses much faster than the other. At extreme, one thread may not even get scheduled to start running before another terminates. We can use EnforceMOP to specify and enforce a simple thread scheduling fairness property that is less restrictive than deterministic alternate execution, but still avoids starvation: as soon as one thread finishes an iteration of its loop, it gets blocked waiting for the other thread to also finish an iteration of its loop; once both threads complete their loop iteration, one of the threads will be (non-deterministically) allowed to start its next loop iteration, and so on. We can specify this property in EnforceMOP using an FSM, as shown in Figures 3.16 and 3.17. When one thread finishes executing one iteration of `doWork` (in state `OneDone` or `TwoDone`), it waits the other thread to finish its execution of `doWork`. After both threads finish, the monitor switches to its `Init` state.

Although this example is purposely simplistic, it shows that EnforceMOP can be used to specify and enforce fair thread scheduling policies. For instance, in a website where each user is served by a thread, it is important to guarantee no user would wait for a long time. With EnforceMOP it is possible to enforce such properties in a modular way, without introducing

any ad-hoc synchronization code in the system.

Enforcing Specific Testing Schedules

Multithreaded programs exhibit different behaviors under different thread schedules. Thus it is vital to have the ability to control thread schedules when performing unit testing. EnforceMOP can also be used as a testing framework to control thread schedules for each unit test. In that case, each property file is associated with some unit test, and serves as a thread schedule specification. In this section we first present our experience with using EnforceMOP as a testing framework to specify schedules in multithreaded unit tests. Then we compare EnforceMOP with several other testing frameworks for multithreaded programs.

Experience

To evaluate the effectiveness of using EnforceMOP as a testing framework, we took existing multithreaded unit tests and translated them to use EnforceMOP. Most of those tests used `sleep` or other ad-hoc synchronization mechanisms to control thread schedules. We first removed all the schedule control statements in those tests, and then wrote one property file for each testing schedule. We took the subject programs used in previous work [46, 78], and in total we translated 185 tests, as shown in Table 3.3.

When using EnforceMOP to specify thread schedules for a given unit test, the event sequences are already known and fixed. So in most cases there's no need to use complex logic formalisms; it is sufficient to only simply state the events to be executed in order. Indeed, we have used ERE in most of the cases, since the event sequences in a schedule is trivially an ERE expression. In some other cases, we have used PTLTL to specify schedules. PTLTL can be used to check whether a condition about past holds when a new event occurs, so it is suitable for enforcing the order of events.

Although most unit tests are quite simple, there are still cases where one event may occur many times. EnforceMOP is able to deal with repeating events. For example, making use of the * and + ERE constructs, properties can be expressed where an event can occur

multiple times. More details on how EnforceMOP handles repeating events are mentioned in the comparison with IMUnit in Section 3.2.3.

Comparison with IMUnit

IMUnit [46] is a framework used to specify and control thread schedules in multithreaded unit tests. An event in IMUnit is fired explicitly by inserting a method call in the test code. A schedule in IMUnit is given as an annotation within a unit test, and it consists of several orderings between events. For example, `a -> b` specifies event `b` should only happen when event `a` has already happened. We compare EnforceMOP with IMUnit in the following aspects.

Expressiveness

IMUnit is also built upon JavaMOP, but its underlying schedule logic is a fragment of PTLTL which does not support repeating events. Consider the same example in Figure 3.7. With IMUnit we can insert events around the `put` method call, but since the method call is made inside a loop we cannot specify in the test schedule the exact number of occurrences of an event. As already shown in Figure 3.8 with the help of operator + in ERE, EnforceMOP is able to express such schedules. The paper [46] mentioned that there were a few more tests where IMUnit was not able to express the schedules because of repeating events. We have successfully used EnforceMOP to specify and enforce the desired test schedules for all those cases. In fact, in our previous examples for specifying general properties, many events are repeating events and can happen anywhere in the program. Unlike IMUnit, EnforceMOP does not use the exact code location to specify an event; instead, it uses pointcuts to match for events. This way, EnforceMOP supports repeating events as long as the chosen logic plugin supports them.

Performance

The performance of IMUnit was evaluated by comparing the time to run all tests with the time to run all the original sleep base tests. Since most of the sleep bases tests are over estimating the time needed for sleep operations, IMUnit tests were able to provide over

3x speed up over the original tests. We repeated the same set of experiments here. We used EnforceMOP to translate all the sleep based tests IMUnit used in experiments and calculated the speedup of using EnforceMOP to enforce schedules versus the original tests. The results are shown in Table 3.4. We are able to achieve same or better speed up with EnforceMOP.

Comparison with MultithreadedTC

MultithreadedTC [78] is another unit testing framework, used to specify schedules in multithreaded tests using *ticks*. In multithreadedTC each test has to be written as a class, and each method in the class contains the code to be executed by a thread in a test. The test schedule is specified in terms of number of *ticks* with respect to a global logical clock. The method `waitForTick` takes a number as an argument, and it will block the current thread until the global clock reaches that number. The global clock will advance when all the threads are blocked.

Although MultithreadedTC is successfully applied on a number of tests [78], it requires users to change the original test a lot. EnforceMOP does not require users to change the original code at all, the schedule specification file (property) is in a separate file, so it is possible to have multiple schedules applied on a same test. Moreover, the schedule in MultithreadedTC is implicitly embedded using ticks. It may be non-trivial to infer a schedule from a MultithreadedTC test for a complicated test case. In terms of functionality, blocking events in MultithreadedTC are also implicit. Threads blocked by MultithreadedTC will be unblocked when all the threads are blocked. This makes it very hard to specify the scenario where one thread waits for another thread to get blocked using MultithreadedTC, while it is easy to do in EnforceMOP (and also in IMUnit).

Comparison with ConAn

ConAn [59] is a framework used to generate test driver code and schedules based on user provided scripts. Similar to MultithreadedTC, ConAn also employs *ticks* to specify logical time in thread schedules. A test in ConAn consists of a set of `#tick` blocks. Inside each

Subject	Tests
Collections [2]	11
Hadoop [4]	1
JBoss-Cache [50]	20
Lucene [5]	2
Mina [6]	1
Pool [3]	2
Sysunit [25]	10
JSR-166 TCK [49]	138
$\sum$	185

Table 3.3: Subject Programs Statistics

Subject	Original	EnforceMOP	Speedup
Collections	2.22	0.26	8.54
Hadoop	1.39	0.38	3.66
JBoss-Cache	73.07	38.89	1.88
Lucene	10.78	2.87	3.76
Mina	0.24	0.14	1.71
Pool	1.48	0.076	19.47
Sysunit	14.47	0.30	48.23
JSR-166 TCK	16.67	6.48	2.57
GeometricMean			5.56

Table 3.4: Test execution time (s)

`#tick` block there are a number of `#thread` blocks. Each `#thread` block contains the code that will be executed by a thread, and ConAn will generate tests based on that.

Since ConAn is also based on ticks, it also suffers from understandability when the schedule to be specified becomes complicated. Moreover, ConAn does not support blocking events. Ticks in ConAn advance automatically after a fixed amount of time, making it unable to express certain schedules EnforceMOP and other frameworks are able to express.

3.2.4 Discussion

EnforceMOP is implemented by executing the incoming event one step ahead using a cloned monitor. Depending upon the chosen logic formalism to express properties, it may be possible that "one step lookahead" is not enough and could cause a deadlock, even though the property is enforceable. For example, consider the ERE property `get* put`. When event

`put` happens in one thread, EnforceMOP has no way to know whether event `get` will happen in the future or not (because code reachability is an undecidable problem). Executing event `put` as soon as it occurs will not violate the property, but if event `get` happens afterward then the monitor will deadlock because event sequence `put get` violates the property. This deadlock can be avoided if EnforceMOP blocked the thread executing event `put` until all the occurrences of event `get` have happened. To achieve this, an *exploration* capability of EnforceMOP would be desirable. Whenever an event arrives and either blocking the current thread or not will not violate any property, record the current program location as a *choice point* and make a choice about whether to block current thread. After the current execution finishes, re-execute the program from the beginning until reaching the previous choice point, and make a different choice. This way all the possible event sequences will be enumerated so it can be checked which event sequences will obey all the properties without causing deadlocks.

In our experience of using EnforceMOP so far, we have not seen many cases where exploration would be needed. Consequently, we leave it as future work to be investigated if we see more scenarios where exploration would be useful.

Chapter 4

Efficient State-Space Exploration

In this chapter we present techniques that help state-space exploration for multithreaded programs. We first present the ReEx framework and our various exploration strategies developed for the ReEx framework. After that, we present the RV-Causal framework, an exploration strategy that employs maximal causal model during state-space exploration. We compare RV-Causal with existing techniques on several read world multithreaded programs and show our improvement.

Note that the contributions presented in this chapter has already been published in the form of a conference paper and a technical report. ReEx and CAPP has been published at the International Symposium on Software Testing and Analysis 2011 (ISSTA 2011) [48], RV-Causal has been published as a technical report at UIUC [60]. The author of this dissertation is the main author or co-author of above papers, and would like to thank anonymous reviewers for valuable feedback.

4.1 Stateless State-Space Exploration with ReEx

4.1.1 Introduction

As introduced in Chapter 2, the ReEx framework is a stateless exploration tool for multithreaded Java programs. It uses dynamic bytecode instrumentation to manipulate the original program, so that it can systematically re-execute the program with different thread schedules. During the instrumentation, all the synchronization operations and shared fields

```
1  // Schedule prefix for the current execution
2  List⟨String⟩ currentSchedulePrefix;
3
4  // Exploration queue
5  Queue⟨List⟨String⟩⟩ toExplore;
6
7  // Exploration strategy
8  Strategy strategy;
9
10 // Current choice index
11 int currentChoiceIndex;
12
13 // Get schedule prefix before the next re−execution
14 void startExecution() {
15   if (toExplore != ∅) {
16     currentSchedulePrefix = toExplore.poll();
17   } else {
18     return;
19   }
20   currentChoiceIndex = 0;
21 }
22
23 // Execute transitions at each choice point
24 void executeTransition() {
25   if (|currentSchedulePrefix| > currentChoiceIndex) {
26     pickThread(currentSchedulePrefix.get(currentChoiceIndex));
27   } else {
28     strategy.chooseThread(currentSchedulePrefix, toExplore);
29   }
30   ++currentChoiceInde;
31 }
```

Figure 4.1: Exploration Algorithm

access operations will be instrumented by ReEx, so their execution orders can be controlled. The re-execution process will finish if a bug is found (an exception is thrown or a deadlock is found), or if ReEx finishes enumerating all the possible thread schedules. ReEx is an open source tool [79] and it has been used extensively in our early work [48].

Because of the large number of possible thread schedules in multithreaded programs, it is usually infeasible to execute all the thread schedules. Therefore, ReEx employs various *exploration strategies* to select or prioritize schedules during state-space exploration. Figure 4.1 shows the exploration algorithm for ReEx.

On line 2, there is a global schedule prefix for each execution. It is used for ReEx to replay the program up to the previous choice point to avoid duplicate execution. The schedule prefix

is generated by the exploration strategy and it is also stored in a global queue `toExplore` on line 5. There is also a global variable for the choice point index which indicates where the current execution is at. In the beginning of each execution, a new schedule prefix is fetched from the head of the `toExplore` queue, and the choice point index is reset. During the execution, whenever a choice point is reached (encountering a synchronization event or shared variable access event), `executeTransition` on line 23 will be called. If the current choice point index is smaller than the size of the schedule prefix, then the execution will pick the thread according to the schedule prefix. Otherwise, ReEx will let the underlying exploration strategy to decide which thread to choose on line 28. The exploration strategy will not only select threads for the current execution, but it would also modify `toExplore` to prioritize/insert new schedule prefixes generated from the current execution. Those schedule prefixes will be used later on during the exploration.

From the above algorithm we can easily see that the exploration strategy decides how ReEx would explore the entire state space. It can also decide which part of the state space will be explored and which part will be ignored. In the rest of this section we will introduce a few exploration strategies we have developed in the ReEx framework for efficient state-space exploration.

4.1.2 Exploration Strategy

Depth/Breadth First Strategies

Depth/Breadth first strategies are the basic exploration strategies in the ReEx framework. They both explore the *entire* state space without any reduction, meaning that all the possible thread schedules will be executed. Depth first strategy always chooses the current thread to execute whenever it is possible to do so. When a choice point is reached, it first checks what are all the enabled threads. If the thread that executes the last transition is still active, it will choose that thread to execute and put all the other choices into `toExplore` queue for

future exploration. If the last thread is not active, it will randomly choose another thread to execute. Breadth first strategy, on the other hand, will first choose the other threads to execute whenever it is possible.

Depth/Breadth first strategies are easy to understand and implement, and they will eventually cover the entire state space. However, in many cases it is impractical to finish the exploration of the entire state space. For detecting concurrency bugs, depth/breadth first strategies also cannot hit the bugs as fast as other exploration strategies would do. Therefore we use them as a comparison baseline in our evaluation.

Iterative Context Bounding Strategy

Our second exploration strategy in the ReEx framework is called iterative context bounding strategy. It is inspired by the idea from Chess [67], so we also call it Chess strategy in the rest of this dissertation. Chess is based on a critical finding for concurrency bugs, that most concurrency bugs can be exposed within a small number of preemptions. Therefore, it is useful to first explore schedules with fewer preemptions, since they may expose the bugs faster without the need to explore the entire state space.

Chess prioritizes schedules based on the number of preemptions in them. It first explores schedules without any preemption, that is each thread will continue execution until it finishes or waits for some other actions. Then, it will explores schedules with one preemption, two preemptions, and etc. In most settings two preemptions are enough to expose most concurrency bugs.

In our implementation of Chess in the ReEx framework, we uses `Priority Queue` for `toExplore`. During the exploration we maintain a counter for the current preemption limit. At each choice point, we choose the threads that if being chosen, would not exceed the preemption limit. In the meantime, we put the other choices in the corresponding priority queue indexed by their preemption numbers. When the queue of the current preemption limit is empty, we increase the preemption limit and get the queue with the corresponding

preemption limit.

Chess strategy performs very well for finding concurrency bugs, as shown by other researchers [67]. Also Chess is a schedule *prioritization* technique and it is orthogonal to other exploration strategies. Therefore, Chess can be combined with other strategies, making it more appealing for detecting concurrency bugs.

Change-Aware Preemption Prioritization Strategy

Most software bugs, including concurrency bugs, are introduced by incremental changes to the software. Regression testing is the most widely practiced method for ensuring the validity of evolving software. Regression testing involves re-executing the tests for a program when its code changes to ensure that the changes have not introduced a fault that causes test failures. As programs evolve and grow, their test suites also grow, and over time it becomes expensive to re-execute all the tests. The problem is exacerbated when test suites contain multithreaded tests that are generally long running. Therefore, many researches have been done to find bugs faster for evolving software, such as test selection, prioritization and minimization.

Our another exploration strategy in ReEx, Change-Aware Preemption Prioritization (CAPP) strategy, is inspired by the ideas from regression testing research. The basic idea for CAPP is to collect change information between two versions of a program and use that information to reduce the cost for exploration. For example, suppose a bug is introduced in the change between version one and version two. CAPP will compute the change information between version one and two, and during the exploration for version two CAPP will use such information to *prioritize* schedules which contain those changes. Schedules that do not touch those changes will be executed later on with low priorities.

CAPP contains two core parts: 1) Collect change information. CAPP organizes changes between program versions as *Impacted Code Points* (ICPs), and it uses different kinds of ICPs based on changed code lines/statements, methods, and classes, and the impact of these

changes onto fields. 2) Prioritize schedules based on various change-aware heuristics. CAPP can use various heuristics to identify and prioritize change-impacted preemptions based on the set of collected ICPs. We will elaborate on these two parts in this section.

Collecting Impacted Code Points

Collecting ICPs is similar to change-impact analysis [74, 80, 84]. However, the goal of collecting ICPs is to identify points that are more likely to affect fault-revealing schedules and hence should be prioritized earlier. Note that we do *not* ensure that the collected ICPs capture the sound or complete impact of changes: CAPP can identify fewer points than really impacted (because CAPP performs prioritization and not selection/pruning, the points not identified will be explored later), and CAPP can identify more points than really impacted (because those points may be helpful in finding an appropriate schedule). Intuitively, our focus is on capturing the impact of the changes on the communication among threads, i.e., capturing the schedule-relevant points in the code. Since concurrency faults are related to synchronization orders and shared-memory accesses, CAPP collects not only directly changed code elements but also their impact on synchronized regions (blocks/methods) and fields (of shared objects).

An ICP is defined as a 4-tuple $\langle C, M, L, F \rangle$, where C is a class name, M is a method name, L is a line number, and F is a field name. An element of the tuple may be $\perp$ to denote a "don't care" value. For example, the ICP $\langle$`org.apache.mina...ProtocolCodecFilter`, `filterWrite()`, 325, $\perp\rangle$ denotes that line 325, which is in the method `filterWrite()` of the class `org.apache.mina...ProtocolCodecFilter`, is impacted by the changes. As another example, the ICP $\langle$`java.util.concurrent.ConcurrentLinkedQueue`, $\perp$, $\perp$, `head`$\rangle$ denotes that the field `head` of the class `java.util.concurrent.ConcurrentLinkedQueue` is impacted by the changes.

Our CAPP implementation utilizes a multi-step process to collect the set of ICPs. First, a diff utility (specifically, the Eclipse JDK structure diff [85]) is used to collect a set of lines that have been changed. This results in a set of ICPs where only the third element, i.e., the

line, is specified. Then four analyses are performed on the abstract syntax tree (AST) of the changed code to fill in the missing elements of the partial ICPs and add additional ICPs.

First, any partial ICPs with changed lines that affect a synchronized region (e.g., adding the `synchronized` keyword to methods, changing the scope of a `synchronized` block, etc.) are expanded to include the entire region (method/block).

Second, for each partial ICP, the method and class that contain the changed lines are identified and filled into the partial ICP. This is straightforward except for some special cases such as inner or anonymous classes.

Third, for any field accesses (reads or writes) within impacted lines, additional ICPs are added that specify change-impacted fields. For example, if the changed code has an access `o.f` for some object `o` of type `C`, an ICP $\langle$`C`, $\perp$, $\perp$, `f`$\rangle$ is added. Note that this ICP includes no (changed) lines. Indeed, it encodes that *any* access to the field is potentially relevant and not only the accesses within the changed code.

Fourth, additional change-impacted field ICPs are collected by determining the read- and write-sets [81] of all methods that are *directly* invoked from the impacted lines, and using fields from these sets. In case of dynamic dispatch, our implementation does not compute any precise call graph but simply approximates the set of callees using the statically declared type of the receiver objects.

Exploration with Change-Aware Heuristics

The exploration algorithm of CAPP is also based on the algorithm described in Figure 4.1. At each choice point during exploration, CAPP will decide whether the next transition of the thread matches ICPs or not. If it matches, then the schedule with the transition will be selected to execute; otherwise it will be put in `toExplore` queue for later execution. The key question is how to decide whether a transition matches collected ICPs or not. CAPP uses a family of heuristics and each heuristic has a different *matching mode*. Next we introduce all those heuristics in details.

Each heuristic takes two parameters, the prioritization mode and the ICP match mode.

The prioritization mode can be `BASIC` (no prioritization), `ALL`, or `SOME`. It stipulates the conditions under which enabled transitions are kept for the current iteration (or postponed for the next iteration):

`ALL (A)` keeps all of the transitions if they are *all* executing a change-impacted point in the code (as determined by the ICPs). Otherwise, if one or more transitions are not executing a change-impacted point, only one of them is kept. The intuition behind this mode is to prioritize preemptions *only among* threads that are executing change-impacted code, and to force threads that are not executing change-impacted code to reach change-impacted areas (or become disabled).

`SOME (S)` keeps all of the transitions if there *exists* at least one transition in the set that is executing a change-impacted point in the code. Otherwise, if no transition is executing a change-impacted point, only the transition of the currently executing thread is kept. The intuition behind this mode is to prioritize preemptions *between threads* that are executing change-impacted code and other threads that are not.

Note that both modes perform prioritization only if a preemption is possible. If a preemption is not possible, all the enabled transitions are returned. Also note that the prioritization mode relies on the ICP match mode to decide which transitions/threads are executing change-impacted code.

There are seven ICP match modes that determine whether a transition is executing changed-impacted code, based on the `impactedCodePoints` set of collected ICPs. The match modes compare the program counter (i.e., the currently executing line/statement that belongs to some method in some class) and potentially stack trace (which has a number of program counters based on the call chain) of a transition/thread being executed with the collected ICPs:

`CLASS (C)` checks if the class of the program counter matches the class of an ICP.

`CLASS_ON_STACK (CO)` checks if the stack trace contains a class specified in an ICP.

`METHOD (M)` checks if the method of the program counter matches a method specified in an ICP.

`METHOD_ON_STACK (MO)` checks if the stack trace contains a method specified in an ICP.

`LINE (L)` checks if the line matches a line specified in an ICP.

`LINE_ON_STACK (LO)` checks if the stack trace contains a line specified in an ICP.

`FIELD (F)` checks if a field being accessed at a program counter (if any) matches a field specified in an ICP.

The combination of the two (non-`BASIC`) prioritization modes and seven ICP match modes results in 14 different heuristics with which Change-Aware Preemption Prioritization can be instantiated. We refer to the heuristics using the respective ICP match mode and prioritization mode. For example, LS is the *Line Some* heuristic that uses the `LINE` match mode and the `SOME` prioritization mode, and COA is the *Class On-stack All* heuristic that uses the `CLASS_ON_STACK` match mode and the `ALL` prioritization mode.

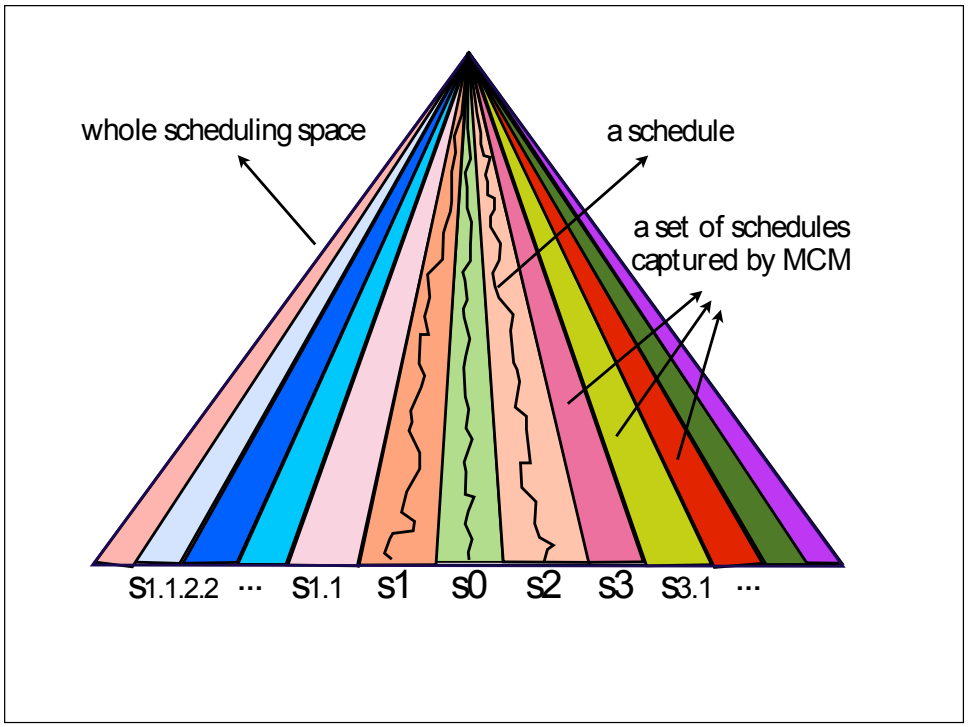
4.2 Systematic Concurrency Testing with Maximal Causality

In this section we introduce RV-Causal, a novel approach for exploration of multithreaded programs using the maximal causal model. It provides reduction of exploration cost without losing the ability to find concurrency bugs. RV-Causal is implemented also as an exploration strategy in ReEx and we also compare RV-Causal with several existing exploration strategies in ReEx.

4.2.1 Motivating Example

Figure 4.2 shows an overview of our approach. Given a certain input, starting with any schedule, our approach systematically covers the entire interleaving space w.r.t. the input by iteratively generating and executing new schedules. In each iteration, RV-Causal takes the trace (an ordered sequence of events) emitted from executing a schedule on the program with our scheduler and monitor, computes a causal set of interleavings corresponding to the schedule according to MCM, and generates new schedules that are not in this causal set. Each causal set is distinct and accounts for a different subspace of the whole scheduling space. To enable checking runtime properties (i.e., safety and liveness properties) over this causal set offline, we encode MCM as a formula of first order logical constraints over a set of order variables (denoting the possible order of each event in the execution), such that any solution to the formula corresponds to a legal interleaving represented by the value of order variables. By encoding the runtime properties as additional constraints and solving a conjunction of the formula and the property constraints with an SMT solver, we can determine whether a property holds or not for all the interleavings in the causal set.

We illustrate our proposed approach using the example in Figure 4.3. Three threads `T1`, `T2` and `T3` are started concurrently, each one has an outer loop with two iterations. `x` and `y` are shared variables among those threads. An error will be triggered in `T3` if (1) `y == 3`



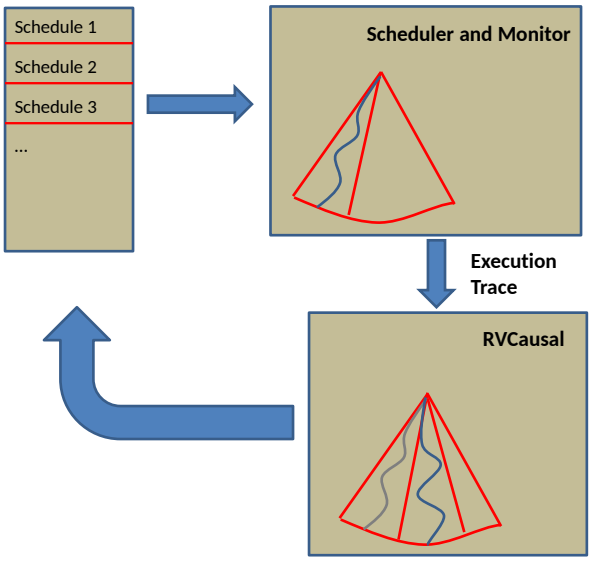


Figure 4.2: Overview

and (2) `x > 1` are both satisfied, as shown at line 12. For (1) to be true, line 9 must be executed after line 14; for (2) to be true, line 2 must be executed between line 7 and line 8.

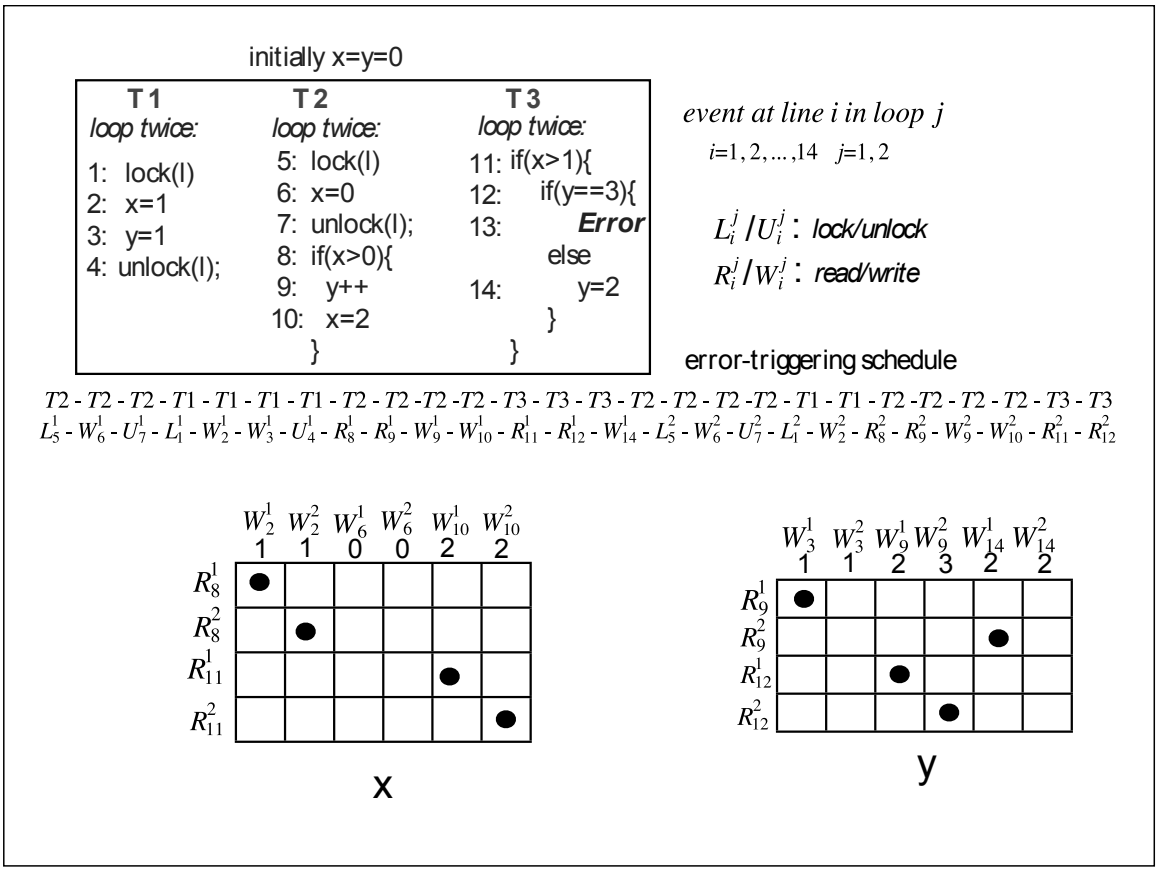


Figure 4.3: Example

The exact buggy schedule is described in Figure 4.3 with four different types of events: R (Read), W (Write), L (Lock) and U (Unlock), and each event is annotated with line number and loop iteration number. For example, R_8^2 means the read event at line 8 in the second loop iteration.

This bug is hard to find by existing state space exploration tools. The reason is that there is a large number of schedules in this program and the bug is hidden deeply with complex thread interleavings. We ran this program with a stateless exploration tool similar to CHESS [68] and used the same iterative context bounding exploration strategy, which is usually the most efficient for finding concurrency bugs. It took 58478 schedules until it hit the bug.

Our approach differs from existing techniques by covering a *set* of schedules from one

single execution. With the maximal causal model [82] as the foundation, our approach is able to analyze an exponential and provably *maximal* number of schedules derived from one single execution trace. From a high level view, our approach works in an iterative manner. In each iteration, we work on one schedule and generate more new schedules, which consists of three functional steps:

Record trace from one execution: We record in this step the necessary information for constructing the maximal causal model. All the reads and writes to shared data will be recorded as well as their value operands. Note that the trace collected here is not required to hit the bug. For example, in the second iteration of `T2`, at line 9 we may read the value 1 for `y`. This will not trigger the error in line 13.

Generate causally different schedules: We use the trace collected in the previous step to construct a maximal causal model. Each maximal causal model contains a set of schedules and our approach will analyze those schedules offline. However, this is still not enough to cover the entire state space. A key novelty of our approach is to systematically generating causally different schedules by forcing reads in the program to *read different values*. For example, suppose we have a trace R_9^2 reads value 1 and W_{14}^1 writes value 2, and R_9^2 happens before W_{14}^1. Our approach will try to force R_9^2 to read a different value, 2 in this case, written by W_{14}^1. All the corresponding constraints will be generated and solved by an SMT solver. If they are satisfiable, such a schedule will be generated.

Note that there may be multiple read operations in the program, so our approach will generate multiple schedules from one input trace. In each generated schedule there will be *at least one* read operation that reads a different value from the original trace, thus guaranteeing that each generated schedule falls into a different causal model.

Figure 4.4 illustrates the schedule generation for our example. `S0` is the schedule in the initial trace. In the first iteration, we generate four new schedules (`S1`, `S2`, `S3`, `S4`), which enforce the four reads (R_8^1, R_8^2 R_{11}^1, and R_{11}^2), respectively, to read value 1. In the second iteration, we continue to work on the traces corresponding to $\mathtt{S}_i$ (i=1,2,3,4 in parallel), and

generate `S1.1`, `S1.2`,...,`S2.1`, `S2.2`, etc. All schedules form a hierarchy, with each child schedule enforcing a different read value. Again, each new schedule may produce a trace containing new read events and/or write values, which can generate new children schedules. For example, our approach will eventually generate the schedule `S1.1.2.2`, which enforces R_{12}^2 to read 3 and triggers the error.

Re-execute program following generated schedules: our generated schedules contain the execution order of threads so they can be used as input for our scheduler to re-execute the program. All the generated schedules will be place into a priority queue. After executing a new schedule in the queue, more causally different schedules may be generated and put in the queue. Our approach will terminate when the queue becomes empty, meaning that there is no more causally different schedule. This is the indication that the entire state space is covered.

4.2.2 Approach

Maximal Causal Model

Our approach builds upon the *maximal causal model* (MCM) foundation, first presented in [82] (for sequential consistency). We briefly review it below.

Multithreaded programs $\mathcal{P}$ are abstracted as the prefix-closed sets of finite traces that they can produce when completely or partially executed, called $\mathcal{P}$*-feasible* traces. A *trace* is abstracted as a sequence of events. *Events* are operations performed by threads on concurrent objects, abstracted as tuples of *attribute-value* pairs. For example, ($thread{=}t_1, op{=}read, target{=}x, data{=}1$) is a read event by thread t_1 to memory location x with value 1. We consider the following common event types:

- $begin(t)/end(t)$: the first/last event of thread t;
- $read(t,x,v)/write(t,x,v)$: read/write a value v on a variable x;

- *lock*(t, l)/*unlock*(t, l): acquire/release a lock l;
- *fork*(t, t'): fork a new thread t';
- *join*(t, t'): block until thread t' terminates;

The sets of $\mathcal{P}$-feasible traces must obey some basic consistency axioms. We proposed two axioms: *prefix closedness* and *local determinism*. The former says that the prefixes of a $\mathcal{P}$-feasible trace are also $\mathcal{P}$-feasible. The latter says that each thread has a deterministic behavior, that is, only the previous events of a thread (and not other events of other threads) determine the next event of the thread, although if that event is a read then it is allowed to get its value from the latest write. These two axioms allow us to associate a causal model *feasible*(τ) to any consistent trace τ, which comprises precisely the traces that can be generated by any program that can generate τ. As shown in [82], *feasible*(τ) is both *sound* and *maximal*: any program which can generate τ can also generate all traces in *feasible*(τ), and for any trace τ' not in *feasible*(τ) there exists a program generating τ which cannot generate τ'. Comparatively, conventional happens-before causal models consisting of all the legal interleavings of τ and their prefixes are *not* maximal [82].

In our approach, we realize MCM using constraints and represent *feasible*(τ) by a formula Φ of first order logic clauses over a set of order variables, each of which corresponds to an event in τ. Any solution to Φ denotes a legal schedule that can produce a corresponding trace in *feasible*(τ). We next describe our constraint modeling.

Constraint Modeling

From a high level view, Φ contains only variables of the form O_e corresponding to events e, which denote the order of the events in a trace in *feasible*(τ). Φ is constructed by a conjunction of three sub-formulas: $\Phi = \Phi_{mhb} \wedge \Phi_{lock} \wedge \Phi_{rw}$.

Must happen-before constraints **(Φ_{mhb})** The must happen-before (MHB) constraints requires that (1) the total orders of the events in each thread are always the same; (2) a

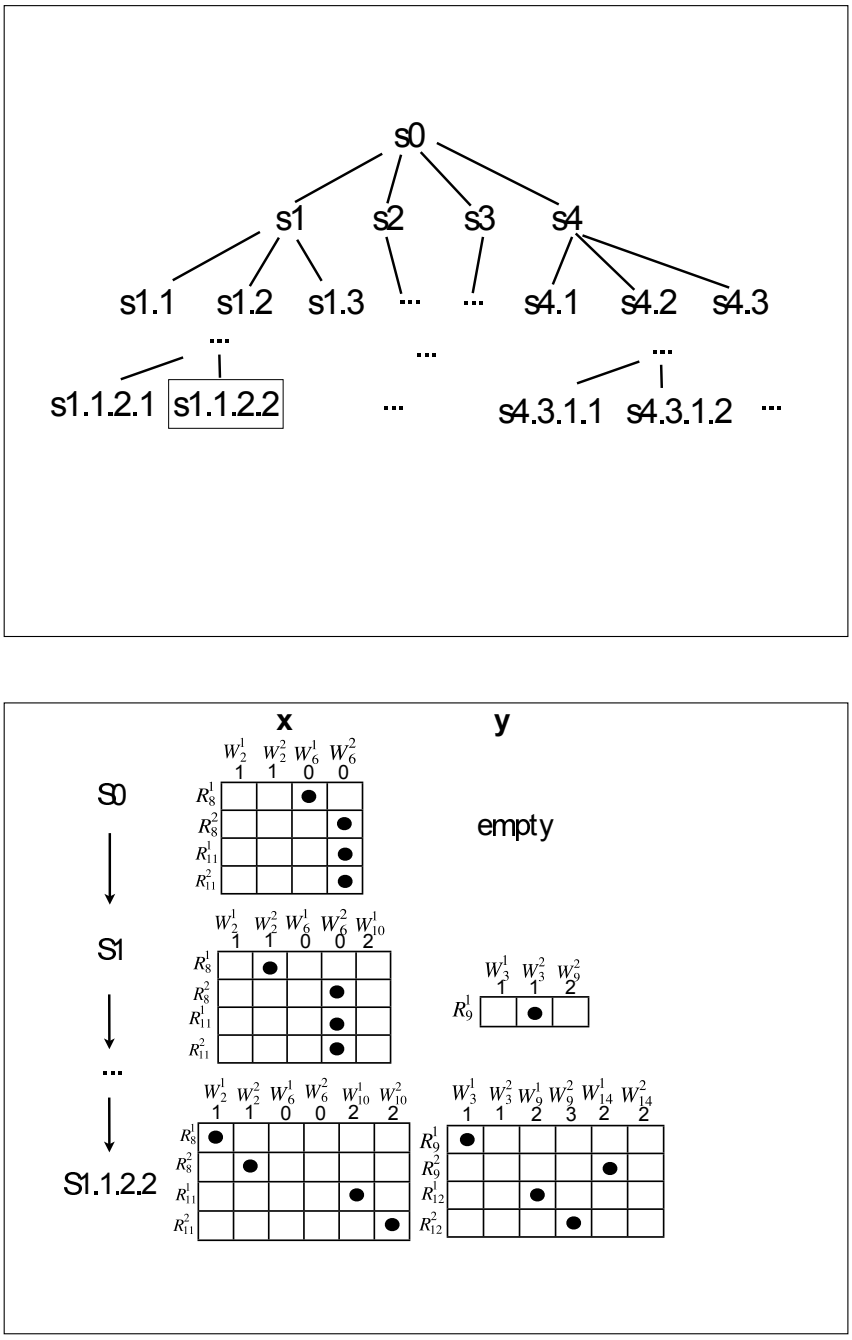


Figure 4.4: Technical overview of schedule generation

begin event can happen only as a first event in a thread and only after the thread is forked by another thread; (3) an *end* event can happen only as the last event in a thread, and a *join* event can happen only after the *end* event of the joined thread. MHB yield an obvious partial order $\prec$ on the events of τ which must be respected by any trace in $\mathit{feasible}(\tau)$. We

can specify $\prec$ easily as constraints over the O variables: we start with $\Phi_{mhb} \equiv \mathit{true}$ and conjunct it with a constraint $O_{e_1} < O_{e_2}$ whenever e_1 and e_2 are events by the same thread and e_1 occurs before e_2, or when e_1 is an event of the form $\mathit{fork}(t, t')$ and e_2 of the form $\mathit{begin}(t')$, etc.

Locking Constraints **(Φ_{lock})** Lock mutual exclusion semantics requires that two sequences of events protected by the same lock do not interleave. Φ_{lock} captures the ordering constraints over the lock *lock* and *unlock* events. For each lock l, we extract the set S_l of all the corresponding pairs (a, r) of *lock*/*unlock* events on l, following the program order locking semantics: the *unlock* is paired with the most recent *lock* on the same lock by the same thread. Then we conjunct Φ_{lock} with the formula

$$\bigwedge_{(a,r),(a',r') \in S_l} (O_r < O_{a'} \vee O_{r'} < O_a)$$

Read-write constraints **(Φ_{rw})** The read-write constraints ensure that every event in the trace is feasible. For an event to be feasible, all the events that must happen-before it should also be feasible. Moreover, any read event that must happen-before it should read the *same value* as that in the original trace. Consider a read event r, say $\mathit{read}(t, x, v)$, we let W^r be the set of $\mathit{write}(\_, x, \_)$ events in τ (here '-' denotes any value), and W^r_v the set of $\mathit{write}(\_, x, v)$ events in τ, then we have the formula defining its feasibility as following:

$$\Phi_{rw}(r) = \bigvee_{w \in W^r_v} (\Phi_{rw}(w) \wedge O_w < O_r \bigwedge_{w \neq w' \in W^r} (O_{w'} < O_w \vee O_r < O_{w'}))$$

The above states that the read event $r = \mathit{read}(t, x, v)$ may read the value v on x written by any *write* event $w = \mathit{write}(\_, x, v)$ in W^r_v (the top disjunction), subject to the condition that the order of w is smaller than that of r and there is no interfering $\mathit{write}(\_, x, \_)$ in between. Moreover, w itself must be concretely feasible, which is ensured by $\Phi_{rw}(w)$. Similarly, $\Phi_{rw}(w)$ is defined by requiring all the reads that must happen-before it are feasible. Φ_{rw} is

a conjunction of $\Phi_{rw}(r)$ for all reads in the considered trace.

Schedule Generation

The goal of schedule generation is to generate schedules that produce traces not in $\mathit{feasible}(\tau)$. Intuitively, this problem is the opposite of constraint modeling in Section 4.2.2 which encodes $\mathit{feasible}(\tau)$. Hence, we can directly leverage the constructed formulae in Φ and negate those that can be negated. Clearly, the only type of such constraints is Φ_{rw}, in which the mapping from read to write may be changed: rather than enforcing a read to read the same value as that in τ, we can instead enforce it to read a different value. The new formula Φ' then encodes a feasible schedule that can produce a different trace (with at least one new event: the read event with a different value). When being re-executed, this new event might change the control flow of the thread, producing more new events. A caveat of this process is that when enforcing a read to read a different value, we must make sure all the reads that must happen before it are matched with writes that write the same values as that in τ. Otherwise, this read event may not be feasible.

Therefore, our algorithm enumerates each read event in τ on the set of all values by the writes on the same variable. For each value that is different from what it reads in τ, we construct Φ' that constrains the read to read the value. We then invoke a constraint solver (such as Z3) to solve Φ'. If the solver returns a solution, the solution represents a new schedule which is feasible and in which the read will read that new value. Note that each read only concerns about the distinct values but not distinct writes. If there are multiple writes writing the same value, it suffices to generate only one new schedule for all of them. This is another salient advantage of our approach: it avoids generating redundant schedules that have the same effect on the program state.

An important property of our algorithm is that it would eventually cover the entire scheduling space.

Proof. (Sketch) For each read, for each new value it can read, our approach generates a new

schedule. Suppose there exists a schedule s not covered, then it must be the case that s contains a new event. There could only be two possibilities for this new event: (1) it is a previously observed event, but reads a new value; (2) it is a previously unseen event. The case (1) is actually impossible, because our algorithm guarantees generating a new schedule for each such read. For (2), it must be the case that the event depends on a branch, the condition of which none of our generated schedules satisfies. However, this means that the branch condition depends on at least one previous read reading a new value, which contradicts to the fact that we already generated one schedule for each read with a different value. □

4.2.3 Implementation

Overflow

Our implementation is on top of ReEx [48], a stateless state-space exploration tool. ReEx is a Java framework used to re-execute multithreaded Java programs based on different exploration strategies. It already has a set of exploration strategies, such as iterative context bounding exploration strategy (Chess) and depth first exploration strategy. We implement our technique as another strategy in ReEx. We use ASM to instrument Java bytecode, such that after each execution all the necessary information is stored in a trace object. In our implemented exploration strategy that trace object serves as input to build constraint model. We solve all the constraints (using Z3) to generate new schedules such that read operation will read a different value. All the new generated schedules will be put in a queue and our exploration strategy will pick the next schedule in the queue to re-execute the program and generate new trace objects. After the queue becomes empty, no new schedule will be generated and the exploration will finish.

Generation of Read Write Matching Pairs

The main part of our implementation is to generate all the possible read write matching pairs, such that the only one read will read a different value, while all other preceding reads read the same values. The algorithm is described in Figure 4.5.

The basic idea of the algorithm is to find all the values a specific read could possible read of, and then recursively generate matching pairs for all those values. Line 5, `getDependentNodes` returns all the nodes that *should happen before* the input node. Those nodes need to appear if the input node will appear in the new schedule. Figure 4.6 describes how we compute dependency nodes. The first case is that two nodes are in the same thread following the program order, then the later node must happen after the early node. In the second case the first thread starts the second thread, then the start node should happen before the very first node from the second thread. In the third case one thread joins on another thread, which means the last node of the joined thread should happen before the join node. In the last case we handle the semantics of `wait-notify` by modeling `wait` operation as `wait` followed by `unlock` and `lock`. Therefore, the wait node should happen before notify node, and notify node should happen before the next node (lock node) following wait node in that thread. By taking into consideration of all the possible *should happen before* relationships in the program, we transitively compute all the dependency nodes for a given node and use that result in the main algorithm.

On line 10, `constructAllReadWritePairs` gets all the values that were written to the same address in previous trace, and calls `constructReadWritePairs` to get the mappings between a read to a specific write node. On line 24, `constructReadWritePairs` takes as input a list of read nodes to be matched, and recursively generates and stores the result in a map. This algorithm will terminate on line 37 and 54 when the list of read nodes is empty. Note that for each read node paired with each possible value it could read, we only need to generate one valid schedule. So we use the global variable `foundSchedule` to terminate the

```
1  // Global variables
2  boolean foundSchedule = false;
3
4  // Get dependent nodes
5  Set⟨AbstractNode⟩ getDependentNodes(Trace trace, AbstractNode node) {
6    // return all the nodes that should happen before the current node
7  }
8
9  // Construct read write pairs
10 void constructAllReadWritePairs(Trace trace, ReadNode targetReadNode) {
11   Set⟨AbstractNode⟩ dependentNodes = getDependentNodes(trace, targetReadNode);
12   Set⟨String⟩ allValues = trace.getAllWriteValuesOnAddr(targetReadNode);
13   for (value ∈ allValues) {
14     if (value == targetReadNode.value) {
15       // match targetReadNode with a different value than the previous trace
16       continue;
17     }
18     foundSchedule = false;
19     constructReadWritePairs(trace, getReadNodes(dependentNodes), dependentNodes,
20       targetReadNode, value, new HashMap⟨ReadNode, AbstractNode⟩());
21   }
22 }
23
24 void constructReadWritePairs(Trace trace, List⟨AbstractNode⟩ readNodes,
25   Set⟨AbstractNode⟩ dependentNodes, ReadNode targetReadNode, String value,
26   Map⟨ReadNode, AbstractNode⟩ readWriteMapping) {
27   if (foundSchedule) {
28     return;
29   }
30   if (readNodes == ∅) {
31     if (readWriteMapping != ∅) {
32       if (canGenerateSchedule(readWriteMapping, targetReadNode, value)) {
33         generateSchedule(readWriteMapping, targetReadNode, value);
34         foundSchedule = true;
35       }
36     }
37     return;
38   }
39
40   ReadNode currentReadNode = removeFirst(readNodes);
41
42   // Already matched or impossible to match currentReadNode
43   while (currentReadNode ∈ readWriteMapping.keySet() || (currentReadNode != targetReadNode
44     && trace.getWriteNodesWithSameValue(currentReadNode) == ∅) {
45     if (readNodes == ∅) {
46       currentReadNode = removeFirst(readNodes);
47     } else {
48       if (readWriteMapping != ∅) {
49         if (canGenerateSchedule(readWriteMapping, targetReadNode, value)) {
50           generateSchedule(readWriteMapping, targetReadNode, value);
51           foundSchedule = true;
52         }
53       }
54       return;
55     }
56   }
57
58   Set⟨WriteNode⟩ writeNodes = trace.getWriteNodesWithSameValue(currentReadNode);
59   for (writeNode ∈ writeNodes) {
60     // Optimizations to prune impossible cases
61     readWriteMapping.put(currentReadNode, writeNode);
62     readNodes.addAll(getReadNodes(getDepedentNodes(trace, writeNode)));
63     dependentNodes.addAll(getDepedentNodes(trace, writeNode));
64     constructReadWritePairs(trace, readNodes, dependentNodes, targetReadNode, value, readWriteMapping);
65   }
66
67   // Handle matching with initial values
68 }
```

Figure 4.5: Generate Read Write Matching Pairs

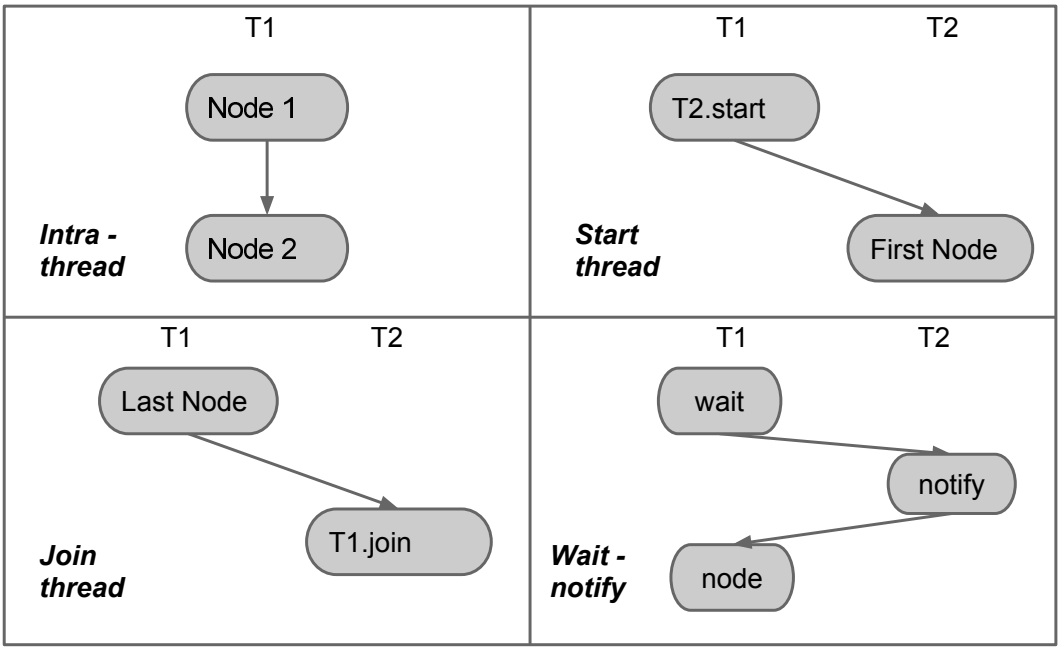


Figure 4.6: Compute dependency nodes

execution earlier in that case for optimization purpose. When the list of read nodes is not empty, the algorithm will get the first read node and pair it with a write node with the same value, put this information in the result map, and recursively execute the algorithm.

Handling matching with initial values:

One problem we encountered during implementation is that for uninitialized variables, JVM will use default values for their data types. Those values are not written by any write nodes, therefore they are not captured in the trace. We handle those nodes in a specific way by using a map to store its initial values. When accessing those variables, we will not only look for existing write nodes in the trace, but also search for the map to find its initial values. For the simplicity purpose, the details of this part is not included in the presented algorithm here.

Optimization:

Table 4.1: Subject Faults and Programs Statistics

	Source	Error
Airline	[29]	Assertion Violation
Account	[29]	Assertion Violation
Allocation	[29]	Assertion Violation
BubbleSort	[29]	Assertion Violation
Lang	[7]	Assertion Violation
Pool	[11]	Assertion Violation
Log4J1	[10]	NullPointer Exception
Log4J2	[8]	NullPointer Exception
Logger	[51]	NullPointer Exception

If we naively pair all the read nodes with possible write nodes in the trace, we may end up with too many possible pairings. Many of those pairings are impossible because of program order constraints. However, we also do not want to check and prune too many impossible pairings in our algorithm, because SMT solver is used to check validity and generate valid schedule. In our implementation, we have done some simple Optimization to quickly check and prune some cases. For example, if there is a cycle between two read write pairs from two different threads, then it is impossible because we are using sequential consistent memory model. Also, if another write nodes that writes a different value to the same address and it should happen before the paired read node and after the paired write node, then it is also impossible for the read node to read the value from the paired write node. The details of those optimization are also omitted on line 60. In practice, those simple optimization give us good performance improvement without losing the benefits of using SMT solver.

4.2.4 Evaluation

Methodology and Subjects

In order to evaluate our technique, we have performed two sets of experiments. First we want to see how RV-Causal would help detecting concurrency bugs. Second, we want to see how RV-Causal would reduce the cost of state-space exploration.

Our subject programs are described in Table 4.1. We collected several concurrency

programs from SIR [29] and also from several large open source systems. Each program has a concurrency bug caused by thread scheduling, associated with a multithreaded unit test which will expose the bug. However, those unit tests will only fail under certain thread schedules. We clarify those bugs into two categories. The first one is `Assertion Violation`, caused by test oracles being violated at the end of the test execution; the second one is `NullPointer Exception`, caused by dereferencing null pointer in test execution.

RV-Causal explores only one schedule in each causal model, and it is useful to find those concurrency bugs because `Assertion Violation` and `NullPointer Exception` are both caused by a read operation matched with a "wrong" operation in one execution. Therefore, it suffices to explore only one schedule in each causal model to reveal those bugs. We think most concurrency bugs fall in those categories except `Deadlock`. We leave that to future work for RV-Causal.

The main objective of our evaluation is to first see how many thread schedules it would take for RV-Causal to hit those bugs, compared with other existing techniques and tools. We then fix those bugs and see how many schedules RV-Causal would take to finish exploring the entire state space. This shows how much state space reduction we can gain by using the maximal causal model.

During our experiments, we compare RV-Causal with DepthFirst Strategy (DFS) and Iterative Context Bounding Strategy [67] (Chess) implemented in ReEx [48]. We choose number of schedules as the metric in our experiments because it was proven to be effective for evaluating state-space exploration techniques in previous work [47].

State-Space Exploration Results

Find concurrency bugs

Table 4.2 summarizes our results of using RV-Causal to find concurrency bugs, compared with DFS and Chess. We set the time limit to be 15 minutes for each subject program, if the exploration does not terminate within that time limit we mark it as `TIMEOUT`.

Table 4.2: Number of Schedules to Find Bugs

	Depth First Search	Chess	RV-Causal
Airline	7	42	3
Account	196	14	4
Allocation	TIMEOUT	15	2
BubbleSort	TIMEOUT	166	9
Lang	TIMEOUT	18	20
Pool	TIMEOUT	128	164
Log4J1	123	8	3
Log4J2	20	29	5
Logger	20	9	3

Overall RV-Causal takes substantially fewer schedules to find bugs than using DFS or Chess exploration strategies. DFS is the basic exploration strategy, which exhaustively enumerates all possible thread schedules one by one until it hits the bugs. Because of the potential large number of possible thread interleavings, DFS cannot finish in time for 5 out of 9 subject programs. Chess is using iterative context bounding approach [67] with bound 2. It only explores schedules with preemptions less or equal to 2, so it does not guarantee to find the bug. However in practice it works very well, as it finds all the bugs in the 9 subject programs. RV-Causal takes even fewer schedules to find those bugs in 7 out of 9 subject programs. That is because many schedules DFS and Chess explores fall into the same causal model, so that RV-Causal will only execute one of them. By forcing a read operation to read a different value in each newly generated schedule, RV-Causal is more likely to lead the program into a new state (e.g., executing a new branch or writing a different value to a shared memory location), thus easier to hit concurrency bugs.

For example, consider the code snippet used in `Allocation` example in Figure 4.7. A few threads are accessing the shared `resultBuf` array concurrently in `run` method. Using DFS or Chess strategy, many context switch points will be created inside those `for` loops on line 2 and 7, even if threads are accessing different elements of the array. RV-Causal will look for the actual dynamic memory locations in each trace and only generate new schedules which lead to different values written to `resultBuf` array, therefore it will not create those unnecessary context switch points.

```
1  void run() {
2      for (int i = 0; i < resultBuf.length; i++) {
3          resultBuf[i] =
4              vector.getFreeBlockAndMarkAsAllocated();
5      }
6
7      for (int i = 0; i < resultBuf.length; i++) {
8          if (resultBuf[i] != -1) {
9              vector.markAsFreeBlock(resultBuf[i]);
10         }
11     }
12 }
13
14 public int getFreeBlockAndMarkAsAllocated() {
15     // bug fix: synchronized (this) {
16         int freeBlockIndex = getFreeBlockIndex();
17         if (freeBlockIndex != -1) {
18             markAsAllocatedBlock(freeBlockIndex);
19         }
20         return freeBlockIndex;
21     // }
22 }
```

Figure 4.7: Allocation example

Explore entire state space

Table 4.3 summarizes our results of using RV-Causal to explore the entire state space on the fixed subject programs. Since all the concurrency bugs are fixed in those subject programs, ReEx will finish exploration only when all the possible thread interleavings are enumerated. Because of the exponential number of thread schedules for multithreaded programs, the naive DFS approach would not be able to finish exploration for 8 out of 9 subject programs. Chess, as a contrast, is able to finish exploration for most subject programs. However, that is due to the fact that Chess only explores thread schedules with preemptions less than 3 among all the possible thread schedules. Therefore, using the Chess approach could possibly miss concurrency bugs (although it was proven to be effective in practice and in our experiments).

RV-Causal takes significantly less schedules to finish exploration in most subject pro-

Table 4.3: Number of Schedules to Finish Exploration

	Depth First Search	Chess	RV-Causal
Airline	TIMEOUT	8309	17
Account	TIMEOUT	819	5
Allocation	TIMEOUT	16311	22
BubbleSort	TIMEOUT	115827	103
Lang	TIMEOUT	9990	334
Pool	TIMEOUT	TIMEOUT	TIMEOUT
Log4J1	329	329	3
Log4J2	TIMEOUT	TIMEOUT	9
Logger	577	138	2

grams, except in POOL where all three approaches could not finish exploration within the time limit. The improvement also comes from the fact that RV-Causal only explores one schedule from each causal model. Consider the example in Figure 4.7 again. The bug was fixed by locking `getFreeBlockAndMarkAsAllocated` method. However Chess and DFS will still explore many alternate interleavings in other methods, resulting in a much larger number of schedules to finish exploration.

Note in some of our subject programs, RV-Causal takes very few schedules to finish exploration. In those programs, developers fixed the bugs by wrapping accesses to shared variables with common locks, or using thread local variables instead of shared variables. In those cases, the total number of causal models decreased significantly compared with those programs before applying their fixes.

4.2.5 Discussion

Comparison with Dynamic Partial Order Reduction

Dynamic Partial Order Reduction (DPOR) is a well known technique for reducing the cost of state-space exploration [35]. The main idea behind DPOR is to look for conflicting and co-enabled transition when program executes. Two transitions are conflicting with each other if at least one of them is a write operation. Whenever two transactions that are accessing the same memory location and are both enabled, a backtrack point will be created to explore

the alternative path.

In [35], the authors presented the following example:

$$T1 : x = 1; x = 2;$$

$$T2 : y = 1; x = 3;$$

T1 and T2 are two different threads executing concurrently. Suppose the first interleaving is $< T1{-}T1{-}T2{-}T2 >$. A backtrack point will be created after executing the first instruction in T1, resulting in interleaving $< T1{-}T2{-}T2{-}T1 >$. Similarly, a backtrack point will be created before executing the first instruction in T1, resulting in interleaving $< T2 - T2 - T1 - T1 >$.

The rationale behind DPOR is that if two transitions are conflicting with each other, then executing them in different orders will lead the program into different states. However, the main difference between DPOR and RV-Causal is that DPOR only looks at all the currently enabled transitions. In other words, it does not take into account the "causal effects" of those transitions. Back into the above example, DPOR would not consider whether there are any read operation that will read those values that being written earlier. Even if there is such a read operation, DPOR would also not consider whether it will read the same value or not. If there is another write operation writes to the same location but with a different value, then all the above interleavings would not show any causally difference.

RV-Causal, on the other hand, looks for interleavings that will result the program fall into another different causal model. Therefore, for a write operation to be considered as a backtrack point, there must be a read operation that reads its value; moreover, it must read a different value than in the previous execution. For example, executing the above program in RV-Causal would only exercise one interleaving, since there is no read operation following those write operations. RV-Causal achieves this goal by modeling the entire program execution and find a viable solution for its causal model. Therefore, RV-Causal is able to further reduce the state space for exploration compared to DPOR.

Deadlock Bugs

Currently our technique tracks each read and write values in the trace and generates different traces such that at least one read will read a new value, due to the definition of the maximal causal model.

Consider a simple example with nested locks: thread T1 acquires lock L1 first and then lock L2, and then it releases L2 followed by L1; thread T2 acquires lock L2 first then lock L1, and then releases L1 followed by L2. After the program finishes executing the first trace (suppose the program does not deadlock in the first trace), RV-Causal will finish exploration because it could not find any new causally different trace. However this program could potentially deadlock if a context switch happens after each thread acquires their first lock.

The reason for our technique to miss this deadlock bug is that when constructing constraints, we only generate *synchronization consistent* traces. That is, we only generate traces in which each lock operation will successfully get the lock. The same goes for wait/notify operations in the program. We currently cannot generate schedules that manifest deadlocks caused by missing notification.

To solve this problem, we need to model lock/unlock and wait/notify operations differently. We will need to model those operations as special kinds of read/write operations and match them with different values in each execution. By doing this our technique will be able to explore schedules that could lead to deadlocks.

Chapter 5

Related Work

5.1 Testing and Runtime Verification of Multithreaded Programs

Most existing work on runtime verification [1, 22, 23, 27, 38, 40, 55, 64] have hardwired specification languages. For example, Java-MaC [55] uses a customized language for interval temporal logic and PaX [40] only supports LTL. Moreover, all existing runtime verification frameworks *monitor* rather than *enforce* properties. JavaMOP [22, 23] is a parametric runtime verification framework which supports multiple logic formalisms. EnforceMOP is extending JavaMOP with the ability to enforce properties in multithreaded programs.

Many approaches have been proposed to test and verify multithreaded programs, such as static/dynamic analysis [16, 33, 34], testing [26, 31, 46, 59, 76–78], and state-space exploration [19, 39, 67]. For enforcing certain schedules in multithreaded code, ConAn [58, 59] and MultithreadedTC [78] introduce unit testing frameworks that allow developers to specify and enforce schedules when writing multithreaded unit tests. ConAn [59] uses a scripting language for specifying method sequences and test schedules to generate test driver code for multithreaded programs. MultithreadedTC [78] employed ticks to specify thread schedules. Our earlier work IMUnit [46] proposed a language with event annotations to specify schedules in multithreaded unit tests. EnforceMOP supports all the features of the above frameworks, as described in Section 3.2.3. Moreover, with the underlying power of various logic formalisms, EnforceMOP can enforce complex schedules precisely and concisely. For finding

bugs in multithreaded code, Falcon [76] and CTrigger [77] employ different mechanisms to improve the probability of context switch and reveal unknown concurrency bugs. Other researchers have also proposed techniques for deterministic record and replay multithreaded programs in order to detect and manifest concurrency bugs [21,43,62]. ConCrash [62] records both thread schedules and method call stack in order to automatically generate unit tests to reproduce concurrency errors after an exception is thrown. Leap [43] is a more recent system which uses local ordering when recording and replaying thread schedules for concurrent programs. Our enforcement and checking mechanism in comparison is targeted towards ensuring the user-specified schedule rather than replaying a previously observed execution. Moreover, EnforceMOP does not aim to find bugs; rather, it is used as a testing framework to specify schedules in multithreaded unit tests.

Our enforcement approach follows the same line of research on automated enforcement of synchronization constraints [14, 15, 20, 28]. Compared with previous approaches, EnforceMOP works on a popular programming language (Java) and supports arbitrary events defined by users. In particular, EnforceMOP currently supports any pointcut that can be captured by AspectJ and it can also be easily extended in future. Moreover, parametric events and various formalisms give EnforceMOP more flexibility to define and enforce synchronization properties.

A data-centric synchronization approach to avoiding certain concurrency errors is proposed in [30, 86]. Their idea is to group fields into atomic sets and automatically enforce the atomicity when accessing those fields at runtime. EnforceMOP follows the same idea of semantic synchronization, but, with its various logic formalisms, EnforceMOP is able to express more complex properties than atomicity. For example, in our evaluation of EnforceMOP, we showed that we can enforce mutual exclusion between a pair of two specific methods; this is a special and finer grained instance of atomicity.

5.2 Efficient State-Space Exploration of Multithreaded Programs

Many work have been proposed for state-space exploration of multithreaded programs. There are typically two ways to explore the state space: *stateful* search and *stateless* search. Stateless search [41, 87] models the state of the program when it executes and use the modeled states to check for errors. For example, Java PathFinder [87] is an explicit state-space exploration tool for checking Java programs. It uses state comparison to do backtracking in its search process. Stateless search [48, 67] does not model the state of the program. Instead, it re-executes the program at all the possible choice points to enumerate all the possible output of program execution. Our work here is built on top of a stateless state-space exploration tool ReEx, however it is possible to extend our work for stateful state-space exploration tools.

Since the entire state space for a multithreaded program is large, it is usually infeasible to explore the whole state space. Researchers have proposed different heuristics to find concurrency bug faster when doing exploration. Chess [67] is built on top of the fact that most concurrency bugs can be found within a small number of preemptions. It then proposes an iterative preemption bounding approach to first explore schedules with a smaller number of preemptions. Following work [13] limits preemptions in a set of selected methods to further improve efficiency of finding concurrency bugs. Wang et el. proposes another heuristic which uses *PSet* coverage information as a guideline when exploring state space [88]. Our earlier work [48] employs a set of heuristics to utilize the change information between program revisions to find concurrency bugs faster. Compared to existing heuristic based work, we do not sacrifice coverage for efficiency when exploring state space. Since our approach is based on maximal causality model, we are able to find bugs faster while being guaranteed to cover all the possible behaviors of multithreaded programs.

There is a rich body of work on using predictive analysis for concurrent programs to find

concurrency bugs, including data races [24, 44], atomicity violations [83] and NullPointer Exceptions [32]. Those techniques differ with one another by the underlying model they are using to represent the program's execution. PENELOPE [83] employs a set of access patterns to synthesize and generate schedules to reveal atomicity violations. Compared to PENELOPE, our work RV-Causal uses the maximal causal model to represent the programs' execution, rather than heuristics-based techniques used in PENELOPE. The maximal causal model guarantees that RV-Causal outperforms existing predictive analysis techniques by the number of feasible schedules inferred from one execution. For data races, jPredictor [24] and RV-Predict [44] both employs sound causal models in their approaches. RV-Predict uses the maximal causal model with control flow to find races, therefore it could find more races than jPredictor and other predictive analysis techniques. RV-Causal also employs the maximal causal model for finding concurrency bugs, but it aims to enumerate all the possible causal models the program can manifest. Therefore, RV-Causal is not sensitive to the input trace, and it can be combined with existing predictive analysis techniques as described in Chapter 6.

Dynamic partial order reduction [35] explores the relationship between enabled transitions during each step of state space exploration. If switching the order of two co-enabled steps does not result in new program state, then it is safer to pick one instead of trying both of them. Our approach tracks the value of each read and write instructions and also takes into account all the constraints when building maximal causality model, which makes our approach subsumes previous partial order reduction work.

Chapter 6

Conclusions and Future Work

Multithreaded programs are hard to develop and test due to the non-deterministic thread scheduler. In this dissertation we present two main bodies of research. First, we present the IMUnit framework for enforcing testing schedules and the EnforceMOP system for enforcing runtime properties. Second, we present the CAPP framework and the RV-Causal framework for efficient state-space exploration of multithreaded programs. We believe our contributions can help developers to develop more reliable multithreaded code. Here we also present directions for potential future work.

Combine RV-Causal with predictive analysis tools:

One motivation of RV-Causal is that we want to augment the effectiveness of existing predictive analysis techniques. Predictive analysis of multithreaded programs takes one trace and predicts concurrency errors based on that one execution. RVPredict [44] is the most recent work based on the maximal causal model. It subsumes all the other existing work by having the maximal predictive power based on one execution. However, RVPredict is still based on *one* trace, so if the input trace does not cover the buggy space, RVPredict will not be able to find the bug. Moreover, running RVPredict multiple times may generate traces that fall into the same causal model, resulting in the same set of errors being reported.

We want to combine our technique with RVPredict and evaluate how many more new bugs can be found by generating traces in another causal model. After RVPredict finishes the prediction for one execution, we want to use RV-Causal to generate new traces. Each trace represents a different causal model, so RVPredict may find completely different set of errors from the new trace. The users of RVPredict can decide whether/when to stop

generating new traces.

Extend RV-Causal to consider deadlock:

Currently RV-Causal tracks the values of all the read and write operations in the trace and generates different traces such that at least one read operation will read a new value. As described in Chapter 4, we cannot handle deadlock bugs at the moment.

The reason for RV-Causal to miss those deadlock bugs is that when we construct constraints, we only generate *synchronization consistent* traces. That is, we only generate traces which guarantee that each lock operation will successfully get the lock, and each wait operation will be successfully notified. We currently cannot generate schedules that manifest deadlock caused by nested locks or missing notification.

To solve this problem, we need to model lock/unlock and wait/notify operations differently. We will need to consider those operations as a special kind of read/write operations and match them with different values in the newly generated execution. By doing this, our technique will be able to generate schedules that could expose those deadlock bugs in multithreaded programs.

Evaluate more extensively and compare with DPOR:

In this dissertation, we compare RV-Causal with DPOR by analyzing a small example to demonstrate the advantages of our technique. Although we have shown clearly that RV-Causal would provide more reduction during state-space exploration, it would be more convincing if we also implement DPOR in the ReEx framework and compare it with RV-Causal on our benchmarks. In general, DPOR is based on the *happens-before* relationship in multithreaded programs, and it was proven before that maximal causal model would subsume happens-before causal model [82]. However, it would still be useful to compare those techniques experimentally.

Explore monitored multithreaded programs:

Another topic for future work is that to combine the JavaMOP framework with state-space exploration tools. JavaMOP currently only works for a single execution of a given

program. Combining JavaMOP with a state-space exploration engine would allow us to systematically explore all the possible executions for a multithreaded program in order to find potential property violations.

Combine runtime verification techniques with testing:

Runtime verification techniques, such as the JavaMOP framework, are very good candidates for being integrated with testing tools. Currently JavaMOP is not widely used in testing practice mainly because 1) it is not very clear for programmers to decide what properties to write and how to write them; 2) the tool itself requires complex set-up. We want to use runtime verification techniques in real world programs to find bugs. We plan to find properties whose violations do not directly lead to exceptions (otherwise those violations will already be caught without using runtime verification techniques), but could potentially lead the program to a bad state. Bugs caused by those properties violations may manifest themselves in a very late stage, and runtime verification techniques can help developers to find them much earlier. In specific, we want to conduct more comprehensive evaluation to see how to use JavaMOP to find new bugs and to help developers to diagnose the root cause of a bug.

Enable runtime verification for message-passing programs:

Currently runtime verification techniques are mostly used for shared-memory programs. We think a possible future research is to use those techniques on message-passing systems. To achieve that, each message in the system needs to be intercepted and analyzed by our techniques. Moreover, we can employ the idea of EnforceMOP to modify, forward or drop malicious messages, in order to enforce correct properties for the entire system. Our initial work on using JavaMOP for robot operating system (ROS) [42] proved this concept on a real message-passing system, therefore we believe this would be a promising direction for future research.

References

[1] Chris Allan, Pavel Avgustinov, Aske Simon Christensen, Laurie J. Hendren, Sascha Kuzins, Ondrej Lhoták, Oege de Moor, Damien Sereni, Ganesh Sittampalam, and Julian Tibble, *Adding trace matching with free variables to AspectJ*, OOPSLA, 2005.

[2] Apache Software Foundation, *Apache Commons Collections*, `http://commons.apache.org/collections/`.

[3] Apache Software Foundation, *Apache Commons Pool*, `http://commons.apache.org/pool/`.

[4] Apache Software Foundation, *Apache Hadoop*, `http://hadoop.apache.org/`.

[5] Apache Software Foundation, *Apache Lucene*, `http://lucene.apache.org/`.

[6] Apache Software Foundation, *Apache MINA*, `http://mina.apache.org/`.

[7] Apache Software Foundation, *LANG-481*, `https://goo.gl/V5xdN6`.

[8] Apache Software Foundation, *LOG4J-44032*, `https://goo.gl/NehbGh`.

[9] Apache Software Foundation, *LOG4J-50213*, `https://goo.gl/BhBnOq`.

[10] Apache Software Foundation, *LOG4J-509*, `https://goo.gl/rtZZLp`.

[11] Apache Software Foundation, *POOL-120*, `https://goo.gl/jsr1nN`.

[12] Apache Software Foundation, *TOMCAT-25841*, `https://goo.gl/RYgi4f`.

[13] Thomas Ball, Sebastian Burckhardt, Katherine Coons, Madanlal Musuvathi, and Shaz Qadeer, *Preemption sealing for efficient concurrency testing*, TACAS, 2010.

[14] Reimer Behrends and R. E. Kurt Stirewalt, *The universe model: an approach for improving the modularity and reliability of concurrent programs*, FSE, 2000.

[15] Aysu Betin-Can and Tevfik Bultan, *Verifiable concurrent programming using concurrency controllers*, ASE, 2004.

[16] Eric Bodden and Klaus Havelund, *Racer: Effective race detection using AspectJ*, ISSTA, 2008.

[17] Sebastian Burckhardt, Chris Dern, Madanlal Musuvathi, and Roy Tan, *Line-Up: A Complete and Automatic Linearizability Checker*, PLDI, 2010.

[18] Sebastian Burckhardt, Pravesh Kothari, Madanlal Musuvathi, and Santosh Nagarakatte, *A randomized scheduler with probabilistic guarantees of finding bugs*, ASPLOS, 2010.

[19] Jacob Burnim, Tayfun Elmas, George Necula, and Koushik Sen, *CONCURRIT: Testing concurrent programs with programmable state-space exploration*, HotPar, 2012.

[20] Roy H. Campbell and A. Nico Habermann, *The specification of process synchronization by path expressions*, OS, 1974.

[21] Richard H. Carver and Kou-Chung Tai, *Replay and testing for concurrent programs*, IEEE Software (1991).

[22] Feng Chen and Grigore Roşu, *Java-MOP: A monitoring oriented programming environment for java*, TACAS, 2005.

[23] Feng Chen and Grigore Roşu, *MOP: An efficient and generic runtime verification framework*, OOPSLA, 2007.

[24] Feng Chen, Traian Florin Şerbănuţă, and Grigore Roşu, *jPredictor: a predictive runtime analysis tool for Java*, ICSE, 2008.

[25] Codehaus, *Sysunit*, `http://docs.codehaus.org/display/SYSUNIT/Home`.

[26] Katherine Coons, Sebastian Burckhardt, and Madanlal Musuvathi, *Gambit: Effective Unit Testing for Concurrency Libraries*, PPoPP, 2010.

[27] Marcelo d'Amorim and Klaus Havelund, *Event-based runtime verification of java programs*, WODA, 2005.

[28] Xianghua Deng, Matthew B. Dwyer, John Hatcliff, and Masaaki Mizuno, *Invariant-based specification, synthesis, and verification of synchronization in concurrent programs*, ICSE, 2002.

[29] Hyunsook Do, Sebastian G. Elbaum, and Gregg Rothermel, *Supporting controlled experimentation with testing techniques: An infrastructure and its potential impact.*, Springer ESE (2005).

[30] Julian Dolby, Christian Hammer, Daniel Marino, Frank Tip, Mandana Vaziri, and Jan Vitek, *A data-centric approach to synchronization*, ACM TOPLAS (2012).

[31] Orit Edelstein, Eitan Farchi, Evgeny Goldin, Yarden Nir, Gil Ratsaby, and Shmuel Ur, *Framework for Testing Multi-Threaded Java Programs*, Wiley CCPE (2003).

[32] Azadeh Farzan, P. Madhusudan, Niloofar Razavi, and Francesco Sorrentino, *Predicting null-pointer dereferences in concurrent programs*, FSE, 2012.

[33] Cormac Flanagan and Stephen N. Freund, *Fasttrack: Efficient and precise dynamic race detection*, PLDI, 2009.

[34] Cormac Flanagan, Stephen N. Freund, and Jaeheon Yi, *Velodrome: A sound and complete dynamic atomicity checker for multithreaded programs*, PLDI, 2008.

[35] Cormac Flanagan and Patrice Godefroid, *Dynamic partial-order reduction for model checking software*, POPL, 2005.

[36] Milos Gligoric, Vilas Jagannath, and Darko Marinov, *MuTMuT: Efficient exploration for mutation testing of multithreaded code*, ICST, 2010.

[37] Patrice Godefroid, *Partial-Order methods for the verification of concurrent systems - an approach to the state-explosion problem*, Springer LNCS, 1996.

[38] Simon Goldsmith, Robert O'Callahan, and Alexander Aiken, *Relational queries over program traces*, OOPSLA, 2005.

[39] Klaus Havelund and Thomas Pressburger, *Model checking Java programs using Java PathFinder*, Springer STTT (1999).

[40] Klaus Havelund and Grigore Rosu, *An overview of the runtime verification tool Java PathExplorer*, Springer FMSD (2004).

[41] Gerald Holzmann, *The model checker SPIN*, IEEE TSE (1997).

[42] Jeff Huang, Cansu Erdogan, Yi Zhang, Brandon M. Moore, Qingzhou Luo, Aravind Sundaresan, and Grigore Rosu, *ROSRV: runtime verification for robots*, RV, 2014.

[43] Jeff Huang, Peng Liu, and Charles Zhang, *Leap: lightweight deterministic multi-processor replay of concurrent Java programs*, FSE, 2010.

[44] Jeff Huang, Patrick Meredith, and Grigore Rosu, *Maximal sound predictive race detection with control flow abstraction*, PLDI, 2014.

[45] IBM, *ECLIPSE-369251*, `https://bugs.eclipse.org/bugs/show_bug.cgi?id=369251`.

[46] Vilas Jagannath, Milos Gligoric, Dongyun Jin, Qingzhou Luo, Grigore Rosu, and Darko Marinov, *Improved multithreaded unit testing*, FSE, 2011.

[47] Vilas Jagannath, Matt Kirn, Yu Lin, and Darko Marinov, *Evaluating machine-independent metrics for state-space exploration*, ICST, 2012.

[48] Vilas Jagannath, Qingzhou Luo, and Darko Marinov, *Change-aware preemption prioritization*, ISSTA, 2011.

[49] Java Community Process, *JSR 166: Concurrency utilities*, `http://g.oswego.edu/dl/concurrency-interest/`.

[50] JBoss Community, *JBoss Cache*, `http://www.jboss.org/jbosscache`.

[51] JDK, *LOGGER-4779253*, `http://goo.gl/qYDwXL`.

[52] Pallavi Joshi, Mayur Naik, and Koushik Sen, *An effective dynamic analysis for detecting generalized deadlocks*, FSE, 2010.

[53] *JPF home page*, `http://babelfish.arc.nasa.gov/trac/jpf/`.

[54] Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William G. Griswold, *An overview of AspectJ*, ECOOP, 2001.

[55] Moonjoo Kim, Sampath Kannan, Insup Lee, Oleg Sokolsky, and Mahesh Viswanathan, *Java-MaC: a run-time assurance tool for Java programs*, Elsevier ENTCS (2001).

[56] Leslie Lamport, *Time, clocks, and the ordering of events in a distributed system*, ACM CACM (1978).

[57] Lassi Project, *Sleep testcase*, `http://tinyurl.com/4hk9zdr`.

[58] Brad Long, Daniel Hoffman, and Paul A. Strooper, *A concurrency test tool for Java monitors*, ASE, 2001.

[59] Brad Long, Daniel Hoffman, and Paul A. Strooper, *Tool support for testing concurrent Java components*, IEEE TSE (2003).

[60] Qingzhou Luo, Jeff Huang, and Grigore Rosu, *Systematic Concurrency Testing with Maximal Causality*, Tech. report, University of Illinois Urbana Champaign, 2015.

[61] Qingzhou Luo and Grigore Rosu, *EnforceMOP: a runtime property enforcement system for multithreaded programs*, ISSTA, 2013.

[62] Qingzhou Luo, Sai Zhang, Jianjun Zhao, and Min Hu, *A lightweight and portable approach to making concurrent failures reproducible*, FASE, 2010.

[63] Qingzhou Luo, Yi Zhang, Choonghwan Lee, Dongyun Jin, Patrick O'Neil Meredith, Traian-Florin Serbanuta, and Grigore Rosu, *Rv-monitor: Efficient parametric runtime verification with simultaneous properties*, RV, 2014.

[64] Michael C. Martin, V. Benjamin Livshits, and Monica S. Lam, *Finding application errors and security flaws using PQL: a program query language*, OOPSLA, 2005.

[65] Patrick O'Neil Meredith, Dongyun Jin, Dennis Griffith, Feng Chen, and Grigore Roşu, *An overview of the MOP runtime verification framework*, Springer STTT (2011).

[66] Madan Musuvathi and Shaz Qadeer, *Chess: systematic stress testing of concurrent software*, LOPSTR, 2006.

[67] Madanlal Musuvathi and Shaz Qadeer, *Iterative context bounding for systematic testing of multithreaded programs*, PLDI, 2007.

[68] Madanlal Musuvathi, Shaz Qadeer, Thomas Ball, Gérard Basler, Piramanayagam A. Nainar, and Iulian Neamtiu, *Finding and reproducing heisenbugs in concurrent programs*, OSDI, 2008.

[69] Santosh Nagarakatte, Sebastian Burckhardt, Milo M.K. Martin, and Madanlal Musuvathi, *Multicore acceleration of priority-based schedulers for concurrency bug detection*, PLDI, 2012.

[70] Mayur Naik, Alex Aiken, and John Whaley, *Effective static race detection for Java*, PLDI, 2006.

[71] Object Refinery, *JFREECHART-1051*, `http://goo.gl/SdkWfC`.

[72] Object Refinery, *JFREECHART-187*, `http://goo.gl/zXrXRO`.

[73] Oracle, *JavaDoc ArrayList*, `http://goo.gl/1aoDol`.

[74] Alessandro Orso, Taweesup Apiwattanapong, and Mary Jean Harrold, *Leveraging field data for impact analysis and regression testing*, ESEC/FSE, 2003.

[75] Chang-Seo Park and Koushik Sen, *Randomized active atomicity violation detection in concurrent programs*, FSE, 2008.

[76] Sangmin Park, Richard W. Vuduc, and Mary Jean Harrold, *Falcon: fault localization in concurrent programs*, ICSE, 2010.

[77] Soyeon Park, Shan Lu, and Yuanyuan Zhou, *CTrigger: Exposing atomicity violation bugs from their hiding places*, ASPLOS, 2009.

[78] William Pugh and Nathaniel Ayewah, *Unit testing concurrent software*, ASE, 2007.

[79] *ReEx home page*, `http://mir.cs.illinois.edu/reex/`.

[80] Xiaoxia Ren, Fenil Shah, Frank Tip, Barbara Ryder, and Ophelia Chesley, *Chianti: A tool for change impact analysis of Java programs*, OOPSLA, 2004.

[81] Atanas Rountev, *Precise identification of side-effect-free methods in Java*, ICSM, 2004.

[82] Traian Florin Serbanuta, Feng Chen, and Grigore Rosu, *Maximal causal models for sequentially consistent systems*, RV, 2012.

[83] Francesco Sorrentino, Azadeh Farzan, and P. Madhusudan, *PENELOPE: weaving threads to expose atomicity violations*, FSE, 2010.

[84] Amitabh Srivastava and Jay Thiagarajan, *Effectively prioritizing tests in development environment*, ISSTA, 2002.

[85] The Eclipse Foundation, *Eclipse JDT UI*, `http://www.eclipse.org/jdt/ui/`.

[86] Mandana Vaziri, Frank Tip, and Julian Dolby, *Associating synchronization constraints with data in an object-oriented language*, POPL, 2006.

[87] Willem Visser, Klaus Havelund, Guillaume P. Brat, Seungjoon Park, and Flavio Lerda, *Model checking programs*, Springer ASE (2003).

[88] Chao Wang, Mahmoud Said, and Aarti Gupta, *Coverage guided systematic concurrency testing*, ICSE, 2011.

[89] Guowei Yang, Matthew B. Dwyer, and Gregg Rothermel, *Regression model checking*, ICSM, 2009.

[90] Shin Yoo and Mark Harman, *Regression testing minimization, selection and prioritization: A survey*, Wiley STVR (2010).

[91] Jie Yu, Satish Narayanasamy, Cristiano Pereira, and Gilles Pokam, *Maple: A coverage-driven testing tool for multithreaded programs*, OOPSLA, 2012.