IMPROVED REGRESSION TESTING OF MULTITHREADED PROGRAMS

BY

VILAS SHEKHAR BANGALORE JAGANNATH

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, 2012

Urbana, Illinois

Doctoral Committee:

Associate Professor Darko Marinov, Chair
Professor Gul Agha
Associate Professor Grigore Roşu
Dr. Michael Hind, Senior Manager at IBM Research

Abstract

The advent of multicore processors has necessitated the use of parallelism to extract greater software performance. Shared-memory multithreaded programming is currently the dominant parallel programming paradigm. However, multithreaded programs are difficult to get right and are often afflicted by bugs like data races, deadlocks, and atomicity violations which may be triggered only by a specific set of schedules.

Multithreaded programs are also difficult to test. Since the behavior of multithreaded programs can depend on the schedule, developers need to express and enforce schedules in multithreaded tests. However, there exists no reliable, modular, efficient, and intuitive methodology for expressing and enforcing schedules in multithreaded tests. Traditionally, developers enforce schedules with time delays, e.g., using Thread.sleep in Java. Unfortunately, this sleep-based approach can produce false positives or negatives, and can result in unnecessarily long testing time.

This dissertation presents a novel framework, called IMUnit, for expressing and enforcing schedules reliably and efficiently in multithreaded tests. IMUnit includes a new language for specifying schedules as constraints on events encountered during test execution and a tool for automatically enforcing the specified schedules during test execution. This dissertation also introduces a tool that helps developers migrate their legacy, sleep-based tests into event-based IMUnit tests. The tool uses new techniques for inferring events and schedules from the executions of sleep-based tests. The inference techniques have high precision and recall, of over 75%, and compared to sleep-based tests, IMUnit reduces testing time on average 3.39x. We also describe our experience in migrating over 200 sleep-based tests.

Since each multithreaded test can have different results for different schedules, it needs to be explored for multiple schedules (ideally all possible schedules) to ensure the property being tested. Exploration is expensive, especially in the context of regression testing where tests need to be re-explored when programs evolve. Most recent research on testing multithreaded code focuses on improving the exploration for one code version. While there have been promising results, most techniques are slow and do not exploit the fact that code evolves.

To improve the exploration of multithreaded tests in the regression testing context, this dissertation proposes a technique, called CAPP, that leverages knowledge about code evolution to prioritize the exploration of multithreaded tests. We evaluated CAPP on the detection of 15 faults in multithreaded Java programs, including large open-source programs, and found that the technique can significantly reduce the exploration required to detect regression faults in multithreaded code compared to the state-of-the-art exploration techniques that do not prioritize exploration based on code evolution.

To my family

Acknowledgments

No PhD is accomplished alone. I have many to thank for their part in mine. No amount of thanks will do them justice, but I would like to thank them anyway, especially:

- My parents, my sister, and Diana for their love and support.
- Prof. Darko Marinov for his unbounded dedication and drive, guidance, advice, and support.
- Prof. Gul Agha for introducing me to graduate research, and for his feedback and continued support.
- Prof. Grigore Rosu for his collaboration, feedback, and support.
- Dr. Mike Hind for his feedback.
- Dr. Mihai Budiu for his valuable guidance, help, and support.
- Milos Gligoric, Qingzhou Luo, and Dongyun Jin for their vital collaboration towards the work presented in this dissertation.
- Other collaborators during my PhD including Brett Daniel, Rohan Sharma, Matt Kirn, Yu Lin, Marcelo d'Amorim, Steven Lauterburg, Zuoning Yin, Johnston Jiaa, Shin Hwei Tan, Tihomir Gvero, Sarfraz Khurshid, Viktor Kuncak, Danny Dig, Elton Alves, Damion Mitchell, and Jurand Nogiec.
- Other colleagues and friends at Illinois including Kevin Lin, Lucas Cook, Alejandro Gutierrez, Yun Young Lee, Nathan Hirtz, Jeff Overby, Vijay Anand Reddy, Nicholas

Chen, Stas Negara, Rajesh Karmani, and Adrian Nistor for making my graduate student life fun, interesting and productive.

I sincerely apologize to anyone I may have inadvertently omitted, and thank them for their help.

Table of Contents

List of Tables

List of Figures

List of Abbreviations

AST	Abstract Syntax Tree
CAPP	Change-Aware Preemption Prioritization
CA	Class All
ICP	Impacted Code Point
COA	Class On-stack All
COS	Class On-stack Some
CS	Class Some
FA	Field All
FS	Field Some
IMUnit	Improved/Illinois Multithreaded Unit Testing (pronounced *immunity*)
IO	Input Output
JPF	Java PathFinder
JVM	Java Virtual Machine
LA	Line All
LOA	Line On-stack All
LOS	Line On-stack Some
LS	Line Some
MA	Method All
MOA	Method On-stack All
MOS	Method On-stack Some

MS	Method Some
ReEx	Re-Execution based Exploration
SUT	System Under Test
TDD	Test-Driven Development

Chapter 1

Introduction

The advent of multicore processors is ushering in a new era in computing. To extract greater performance from multicore 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. For example, the Java programming language provides support for threads in the language and libraries, with shared data residing on the heap.

Unfortunately, multithreaded programs are notoriously difficult to get right. Due to the non-determinism introduced by thread scheduling, a multithreaded program with fixed inputs can display different behaviors for different thread *schedules*. Since there are typically a large number of possible schedules, it is challenging for developers to reason about all of them. Hence, multithreaded programs are often afflicted by hard to detect faults like data races, atomicity violations, and deadlocks, which are triggered by a small specific set of schedules.

Multithreaded programs are also notoriously difficult to test. Since the correctness of multithreaded programs is predicated by thread schedules, developers need to be able to *express* and *enforce* schedules in multithreaded tests. However, there exists no reliable, modular, efficient, and intuitive methodology for expressing and enforcing schedules in multithreaded tests. Also, since multithreaded tests may have a large number of possible schedules, the tests need to be *explored* for multiple schedules to ensure the property being tested. Exploration of multithreaded tests is resource intensive and even more so in the context of *regression testing*, where tests have to be re-explored whenever a program evolves in order

to ensure that the program has not regressed. This dissertation presents our research dealing with these issues faced during the testing of multithreaded programs. Specifically, we present a framework for expressing and enforcing schedules in multithreaded tests, and a technique for prioritizing the exploration of multithreaded tests in a regression testing scenario.

1.1 Thesis Statement

Our thesis is that:

> *It is possible to (1) build a framework that enables better expression and enforcement of schedules in multithreaded tests and hence helps developers write better multithreaded tests, and (2) develop a technique that improves the exploration of multithreaded tests in a regression testing scenario by utilizing evolution information.*

To confirm this thesis, this dissertation presents two main bodies of research centered around: (1) the IMUnit framework for writing multithreaded tests and (2) the CAPP technique for change-aware exploration of multithreaded tests. The rest of this chapter introduces these two areas of research.

1.2 Writing Multithreaded Tests

To validate their multithreaded code, developers need to 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. Since the correctness of multithreaded programs can be predicated by the schedule, developers often want to express and enforce a particular schedule[1] for a test. For example,

[1]The terms "schedule" and "set of schedules" are used interchangeably in the rest of the dissertation.

consider two threads, one executing a method m and the other executing a method m'. Developers may want to ensure in one test that m finishes on one thread 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 (i.e., expressing and enforcing) the schedule, it is impossible to write precise assertions about the execution because the results can differ in the two scenarios, and also, without controlling the schedule, it is impossible to guarantee which scenarios would be covered during testing, even if multiple testing runs are performed.

To control the schedule in multithreaded tests, in current practice, 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 during the execution of a multithreaded test, and then assert the intended result for the desired schedule. However, sleeps are an unreliable and inefficient mechanism for enforcing schedules because sleeps are based on real time. 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). 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, sleeps can be inaccurate even on a single machine [68]. 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.

Researchers have previously noted the numerous problems with using sleeps to specify schedules in multithreaded tests and have developed frameworks such as ConAn [72, 73], ConcJUnit [90], MultithreadedTC [88], and ThreadControl [34] to tackle some problems in specifying and enforcing schedules in multithreaded unit tests. However, despite these frameworks, multithreaded unit testing still has many issues that could be categorized as follows:

Readability: Most current frameworks force developers to reason about the execution of threads relative to a global clock. This is unintuitive since developers usually reason about the execution of their multithreaded code in terms of event relationships (such as m finishing before m' starts). Some frameworks, like ConAn, require users to write schedules in external scripts, which makes it even more difficult to reason about schedules. In other frameworks the schedule is implicit, as a part of the unit test code, and hence it is difficult to focus on the schedule and reason about it separately at a higher level.

Modularity: In some current frameworks, like MultithreadedTC, as well as the legacy sleep-based tests, the intended schedule is intermixed with the test code and effectively hard-coded into a multithreaded unit test. This makes it difficult to specify multiple schedules for a particular unit test and/or to reuse test code among different tests.

Reliability: Some current frameworks, like ConAn, as well as the legacy sleep-based tests, rely on real time. As explained, this makes them very fragile, leading to false positives/negatives and/or slow testing time.

Migration Costs: Most current frameworks are very different from the traditional sleep-based tests. This makes it costly to migrate the existing sleep-based tests to those frameworks. For example, ConAn requires tests to be written using a special purpose scripting language, and MultithreadedTC requires each test to be written in a separate test class with the code for each thread in a separate, specially named method.

We present a new framework, called *IMUnit*[2] (*pronounced "immunity"*, as it helps to

[2]As per the xUnit naming scheme, IMUnit is derived from "Improved/Illinois Multithreaded Unit testing".

make multithreaded code more "immune" to bugs), which aims to address these issues with multithreaded unit testing. Specifically, we make the following contributions towards improved expression of multithreaded tests:

- **Schedule Language:** 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 [25,77] 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, hence each IMUnit schedule represents a set of schedules that can be enforced during the execution of a test.

  While the ideas of IMUnit can be embodied in any language, we have developed two implementations for Java, IMUnit Original and IMUnit Light. Syntactically, the IMUnit constructs are represented using Java annotations. A developer can use `@Event` and `@Schedule` annotations to describe the events and intended schedules, respectively, for a multithreaded unit test. Note that `@Event` annotations appear on statements. However, Java currently (version 7) does not support annotations on statements. For the time being, `@Event` annotations can be written as comments, e.g., `/* @Event("finishedAdd1") */`, which IMUnit Original translates into code for test execution. Alternatively, in IMUnit Light, events are specified through `fireEvent("eventName")` calls. `@Schedule` annotations appear on methods, so they are already fully supported in the current version of Java.

- **Automated Migration:** We have developed two inference techniques and a tool to

ease migration of legacy, sleep-based tests to IMUnit, event-based tests. Our inference techniques can automatically infer likely relevant events and schedules from the execution traces of existing sleep-based tests. We implemented our migration tool as an Eclipse plugin which uses the results of inference to automatically refactor a given sleep-based multithreaded unit test into an event-based IMUnit test.

- **Execution and Checking:** As mentioned earlier, we have implemented two tools for the execution of IMUnit multithreaded unit tests. The first tool, called IMUnit Original, can execute tests in one of two modes. In the active mode, it controls the thread scheduler to enforce a given IMUnit schedule during test execution. In the passive mode, it checks whether an arbitrary test execution, controlled by the regular JVM thread scheduler, follows a given IMUnit schedule. To enforce/check the schedules, our IMUnit Original tool uses the JavaMOP monitoring framework [25, 77]. The tool also includes a new runner for the standard JUnit testing framework to enable execution of IMUnit tests with our enforcement/checking tool. The runner also detects deadlocks encountered during test execution.

  The second tool, called IMUnit Light, was developed to ease the adoption and use of IMUnit. It provides all the features of the first tool, except support for lesser used language constructs. IMUnit Light has been released publicly along with source code and documentation at `http://mir.cs.illinois.edu/imunit`. It has generated some interest from outside software organizations e.g., developers of the Apache River [10] project have already started using IMUnit to write their multithreaded unit tests.

- **Evaluation:** To guide and refine our design of the IMUnit language, we inspected over 200 sleep-based tests from several open-source projects. We manually translated 198 of those tests into IMUnit, adding events and schedules, and removing sleeps. As a result, the current version of IMUnit is highly expressive, and we were able to remove all sleeps from all but 4 tests.

We evaluated our inference techniques by automatically inferring events/schedules for the original tests that we manually translated. The subprojects on manual translation and automatic inference were performed by different researchers to reduce the direct bias of manual translation on automatic inference. Computing the precision and recall of the automatically inferred events/schedules with respect to the manually translated events/schedules, we find our techniques to be highly effective, with over 75% precision and recall.

We also compared the execution time of the original tests and our translated tests. Because the main goal of IMUnit is to make tests more readable, modular, and reliable, we did not expect IMUnit to run faster. However, IMUnit did reduce the testing time, on average 3.39x, compared to the sleep-based tests, with the sleep duration that the original tests had in the code. As mentioned earlier, these duration values are often over-estimated, especially in older tests that were written for slower machines. In summary, IMUnit not only makes multithreaded unit tests more readable, modular, and reliable than the traditional sleep-based approach, but IMUnit can also make test execution faster.

Note that the contributions of IMUnit have already been published in the form of a conference paper at the joint meeting of the 19th ACM SIGSOFT Symposium on the Foundations of Software Engineering and 13th European Software Engineering Conference (ESEC/FSE 2011) [57]. The author of this dissertation presented this work at the conference. We would like to thank the audience of the talk for their comments and questions which have been incorporated in this dissertation to improve the presentation and provide additional details where required. We would also like to thank the anonymous reviewers who reviewed our paper for their useful comments.

1.3 Exploration of Multithreaded Tests

Ensuring the reliability of multithreaded code has been an active area of research with several promising recent tools and results [20, 22, 29, 42, 43, 64, 78, 80, 84]. Most of these tools execute multithreaded tests to 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, and as a result they tend to be fairly time consuming. 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 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. While many techniques have been proposed to alleviate this problem for sequential tests [108], there is much less work for multithreaded code [50, 51, 106].

Yoo and Harman [108] present a detailed survey of regression testing techniques (mostly for sequential programs) that minimize (e.g., [55]), select (e.g., [52,105]), or prioritize (e.g., [39, 40,62,71,95,104,110]) test suites. Test selection determines which tests to rerun after changing code, and test prioritization determines in what order to run tests to find faults faster. The techniques for sequential code showed good results in practice (e.g., [98]) but unfortunately cannot be applied directly for multithreaded code. Specifically, those techniques do not target exploration of schedules *within* one test, although they can be applied *across* tests.

There is some recent work on targeting program changes in systematic testing for multi-

threaded code [51,106] or sequential code (with explicit non-determinism) [69]. The proposed techniques reuse results from exploration of one program version to speed up exploration of the next program version (or a code mutant). These techniques in effect perform *selection*, pruning from exploration the schedules that are unaffected by the code changes, which is complementary to prioritization. In general, prioritization could be used in conjunction with selection to prioritize already selected parts of the exploration.

We make the following contributions towards improving the exploration of multithreaded regression tests:

- **Technique:** We propose a new technique, called *Change-Aware Preemption Prioritization (CAPP)*, that uses code evolution information 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 prioritizes the order of exploration of thread schedules based on how test exploration dynamically encounters changed code.

- **Heuristics:** CAPP is a general technique that can be instantiated with different definitions of code changes and scheduling choices to prioritize. We present 14 heuristics that consider changes at the level of source-code lines/statements, methods, classes, or fields affected by the change, and consider prioritizing scheduling choices based on whether all or only some executing threads are executing changed code.

- **Implementation:** We have implemented CAPP in two frameworks for systematic exploration of multithreaded Java code. Java PathFinder (JPF) [65, 103] is a widely used tool for checking Java code. JPF performs a stateful search, with checkpointing and restoration of program state to explore thread schedules, and with state comparison to prune the search. ReEx is a tool that we implemented following the ideas from CHESS [78]. ReEx performs a stateless search, with re-execution to establish a program state to explore thread schedules, and with no state comparison.

- **Evaluation:** We evaluated CAPP and its heuristics on the detection of 15 faults in multithreaded Java code, including some large open-source programs. Our evaluation addresses the following research questions:

  RQ1: How much reduction in exploration cost does CAPP provide over change-unaware techniques?

  RQ2: How do the heuristics compare with each other?

  RQ3: How do the results compare across stateful/stateless exploration?

  RQ4: How do the results compare across default/randomized search order?

  In short, the results show that CAPP can substantially reduce the exploration cost required to detect multithreaded regression faults, and there are interesting variations in cost among heuristics, stateful/stateless search, and default/randomized search order.

Note that the contributions of CAPP have already been published in the form of a conference paper at the 20th International Symposium on Software Testing and Analysis (ISSTA 2011). The author of this dissertation presented this work at the conference. We would like to thank the audience of the talk for their comments and questions which have been incorporated in this dissertation to improve the presentation and provide additional details where required. We would also like to thank the anonymous reviewers who reviewed our paper for their useful comments.

1.4 Dissertation Organization

The rest of this dissertation is organized as follows:

Chapter 2: Background This chapter presents a brief overview of regression testing and the exploration of multithreaded tests in order to set the context for the presentation of the contributions of this dissertation in later chapters.

Chapter 3: IMUnit This chapter presents the contributions of the IMUnit framework, which helps developers write better multithreaded unit/regression tests by improving the expression and enforcement of schedules in multithreaded tests.

Chapter 4: CAPP This chapter presents the contributions of the CAPP technique, which we developed to improve the exploration of multithreaded tests in a regression testing scenario.

Chapter 5: Related Work This chapter presents an overview of the various bodies of work that are related to the contributions of this dissertation.

Chapter 6: Conclusion and Future Work This chapter concludes the dissertation and presents various avenues for future work building upon the contributions of this dissertation.

Chapter 2

Background

This chapter presents some of the background necessary to set the context for the contributions of this dissertation. Section 2.1 presents an overview of regression testing, and Section 2.2 describes the complexities involved in testing multithreaded programs.

2.1 Regression Testing

Successful software evolves continuously in response to feature requests, bug reports, design changes, etc. Regression testing [41, 107] is the mostly widely adopted process for ensuring the reliability of evolving software. It requires the maintenance of a regression *test suite*, typically containing many fast running *unit tests*. Each unit test usually exercise a small part (unit) of the code and asserts the intended behavior. When the code evolves, the unit tests are re-executed to detect any regressions caused by the evolution. The faster the regression is detected, the better it is for the developer to reconsider the changes [93]. Also, the unit tests are typically required to achieve certain levels of code coverage in order to provide some confidence in detecting regressions [46].

When a new feature is implemented, unit tests that assert the indented behavior of the feature are added to the regression test suite. If the developers used Test-Driven Development (TDD) [19], such tests would be written before the feature is implemented, and then utilized to guide the implementation of the feature. When a new bug is reported, a unit test is written, which fails in the presence of the bug and would pass if the bug was fixed. This unit test helps the developers during the debugging and bug fixing process. After the bug is

fixed, the test is retained in the regression test suite to ensure the bug is not re-introduced by future evolutions (i.e., the test helps avoid regression). So typically a unit test in a regression test suite is written either to assert the intended behavior of a new feature or to prevent the re-introduction of a fixed bug. In Chapter 3, we present the details of the IMUnit framework, which helps developers write better unit tests for multithreaded programs. The unique characteristics of unit tests for multithreaded programs are discussed in Section 2.2.

As the software evolves, and more features are added and more bugs are fixed, the regression test suite grows. When a regression test suite grows too large, it may become unfeasible to re-execute the entire test suite every time the code evolves. The problem is exacerbated for multithreaded programs, where tests need to be explored rather than just executed. Many techniques have been proposed to tackle this problem for sequential programs. These techniques usually utilize information about the code evolution and/or code coverage of the tests. Test suite *minimization* techniques (e.g., [55]) can be used to eliminate redundant tests while maintaining the quality of the test suite. Test suite *prioritization* techniques (e.g., [40, 62, 71, 95, 110]) can be used to re-order the execution of tests such that regressions are detected faster. Test suite *selection* techniques (e.g., [52,105]) can be used to reduce the number of tests that are executed in response to a particular code evolution. Yoo and Harman [108] present a detailed survey of these techniques. Chapter 4 presents the details of the CAPP technique, which we developed to prioritize the exploration of a single multithreaded unit test in a regression testing scenario. Section 2.2 presents a brief description of why multithreaded unit tests need to be explored, and how exploration can be performed.

2.2 Testing Multithreaded Programs

Shared memory multithreaded programming is currently the dominant paradigm for writing parallel programs that can extract greater performance from multicore processors (which are

```
1  public class Counter {
2    private int count = 0;
3
4    public void increment() {
5      this.count++;
6    }
7
8    public void decrement() {
9      if (count > 0) {
10       this.count--;
11     }
12   }
13
14   public int getCount() {
15     return this.count;
16   }
17 }
```

(a) Counter Class

```
1  public void useCounter() {
2    final Counter ctr = new Counter();
3    Thread incThread = new Thread(
4      new Runnable() {
5        public void run() {
6          ctr.increment();
7        }
8      });
9    incThread.start();
10   ctr.decrement();
11   incThread.join();
12   int count = ctr.getCount();
13   // count can be 0 or 1
14   System.out.println(count);
15 }
```

(b) Multithreaded Use of Counter

Figure 2.1: Non-Determinism Due to Thread Scheduling

becoming increasingly mainstream). In this paradigm, multiple *threads* of control communicate with each other by reading and writing shared objects/memory. This communication introduces non-determinism based on the *scheduling* of the threads. Figure 2.1 demonstrates this non-determinism. Figure 2.1(a) shows a simple implementation of a counter which is initialized to a value of zero. The `increment` method can be used to increment the counter value. The `decrement` method can be used to decrement the counter value, but it has no effect if the counter value is zero. The `getCount` method can be used to obtain the current counter value. Figure 2.1(b) shows a usage/test of this class in a multithreaded setting. First a shared counter instance named `ctr` is initialized. Then a new thread is started which performs an `increment` operation on `ctr` while the main thread performs a `decrement` operation on `ctr` in parallel. At the end, the resulting counter value of `ctr` is stored as `count` and printed out.

The value of `count` is non-deterministic. For example, if the `decrement` is performed before the `increment`, the value of `count` will be one. On the other hand, if the `decrement` is performed after the `increment`, the value of `count` will be zero. This non-determinism is introduced by thread scheduling (i.e., which thread gets to execute when). There are

many more possible schedules other than the two examples already described. For example, the `increment` operation may be started in one thread and then the execution could switch to the `decrement` operation in the other thread before the `increment` is completed (this schedule could be avoided by making the methods that access `count` mutually exclusive via the `synchronized` keyword). Essentially, execution may be switched from one thread to another before any instruction is executed. However, only switches before/after schedule-relevant instructions (e.g., shared data read/write, lock/unlock, etc) can lead to schedules that may have different behaviors. Nevertheless, the number of possible schedules blows up exponentially with the number of schedule-relevant instructions. Switching execution from one thread to another is called a *context-switch*. A context-switch is said to be a *preemptive context-switch* (or *preemption*) if the thread from which execution was switched to another thread is still enabled (threads are disabled when they are waiting to acquire a lock held by another thread).

Since different schedules can lead to different behavior/results, the schedule(s) to be used is an additional parameter in multithreaded tests. This is in addition to the three conceptual parts/parameters of all sequential tests, which are the input, the code invoking SUT with input, and the assertions on the output. Specifying schedule(s) in a multithreaded test requires the ability to express the schedule(s) while writing the test, and then the ability to enforce the expressed schedule(s) while executing the test. The mechanisms currently available to developers for expressing and enforcing schedules in multithreaded tests have many drawbacks. Chapter 3 describes these drawbacks and presents the details of the IMUnit framework, which we developed to overcome the drawbacks and help developers write better tests for multithreaded programs.

When a multithreaded test specifies a set of schedules, the assertions in the test are expected to be satisfied only for the specified set of schedules. On the other hand, if a multithreaded test does not specify a particular set of schedules, it is equivalent to specifying the set of all possible schedules i.e., the assertions in the test are expected to be satisfied for

```
1  // Exploration state
2  class Transition { State state; Thread thread; }
3  Set⟨Transition⟩ toExplore;
4  Set⟨State⟩ explored;
5  // Input
6  Test test;
7  // Exploration
8  PassOrFail explore() {
9    State s_init = initial state for test;
10   toExplore = {Transition(s_init, t) | t ∈ enabledThreads(s_init)};
11   if (STATEFUL) {
12     explored = { s_init };
13   }
14   while (toExplore ≠ {}) {
15     Transition current = pickOne(toExplore);
16     toExplore = toExplore − { current };
17     restore current.state;
18     State s' = execute current.thread on current.state;
19     if (s' ∉ explored) {
20       if (s' is errorState) {
21         return FAIL;
22       }
23       Set⟨Transition⟩ enabled = {Transition(s', t') | t' ∈ enabledThreads(s')};
24       toExplore = toExplore ∪ enabled;
25       if (STATEFUL) {
26         explored = explored ∪ { s' };
27       }
28     }
29   }
30   return PASS;
31 }
```

Figure 2.2: Exploration of a Multithreaded Test

all the possible schedules. In both cases, the assertions in a multithreaded test are expected to be satisfied for potentially multiple schedules. Therefore, it is not sufficient to just execute a multithreaded test once, since a particular execution only follows one possible schedule. In order to ensure that the assertions are satisfied for all the possible/specified schedules, multithreaded tests need to be *explored* for each of those schedules.

Exploration of a multithreaded test is essentially an exploration of the dynamic *state-space* graph for the test. Each node of the graph represents a *state* and each edge represents a *transition* that needs to be executed to move from one state to another. A transition consists of a set of instructions that are to be executed on a particular thread. Figure 2.2 shows the pseudo-code for the generic algorithm that can be used to explore a multithreaded test. The

algorithm uses the `Transition` pair to represent a transition that consists of a `State` and a `Thread` that can be executed in that state. The algorithm takes as input the test to be explored. The main data structure of the algorithm is the set `toExplore` which contains transitions that still need to be explored. Optionally, for *stateful* exploration, the algorithm also maintains the set `explored` which contains states that have already been explored.

The algorithm starts by initializing `toExplore` to the set of enabled transitions of the initial state of the test, and optionally initializing `explored` to a set containing the initial state. The enabled transitions for a particular state comprises one transition for every thread that is enabled in that state. A thread is considered enabled if it has been started, has not yet completed execution, and is not currently blocked (e.g., waiting to acquire a lock that is held by another thread). The main exploration loop (lines 14-29) starts after the initialization and continues as long as `toExplore` is not empty. In each iteration of the main loop, a transition is selected and removed from the `toExplore` set. The state of the selected transition is then reestablished. The state can be reestablished either by *restoring* a stored state, which requires additional memory to store states (but only those states that need to be explored, not necessarily all the states that have been explored), or by *re-executing* code to reach the state, which requires additional time for the re-execution. After reestablishing the state, the thread of the selected transition is executed on the state to obtain the next state, `s'`. If `s'` is an error state, a test failure is reported. If `s'` is not an error state, the algorithm proceeds by obtaining the transitions that are enabled in `s'` and adding them to the `toExplore` set. At this point, if the exploration being performed is *stateful*, `s'` is added to the `explored` set. This ensures that the algorithm does not re-explore states that have already been explored, but this requires additional memory for storing states (or hashes of states) that have already been explored. Alternatively, in *stateless* exploration the `explored` set is not maintained, however additional time may be required for re-exploring states that are encountered more than once. This marks the end of one iteration of the main exploration loop. When the `toExplore` set becomes empty, the main exploration loop terminates and

the test is reported to have passed.

Note that the exploration of a multithreaded test can be performed in different orders. If `toExplore` was a stack, the exploration would become depth-first, and if `toExplore` was a queue, the exploration would become breadth-first. In general, `toExplore` could be a priority queue in which transitions could be prioritized using various heuristics. Researchers have introduced many such heuristics to prioritize the exploration, with the intention of detecting bugs faster. An example of such a heuristic, which is used by the CHESS tool, is iterative context bounding [78], where exploration of transitions that require fewer preemptions is prioritized over exploration of transitions that require more preemptions. Many other techniques introduced by researchers for testing multithreaded programs can conceptually be considered exploration prioritization techniques, including Gambit [29], Contest [38], Preemption Sealing [17], and Active Testing [63,84]. All these techniques are focused on exploring multithreaded tests for one version of code. However, in reality code always evolves, and as part of the regression testing process, tests may need to be explored for each new version of code. Chapter 4 presents the details of CAPP, which we developed to prioritize the exploration of a multithreaded test in a regression testing scenario by leveraging evolution information.

The rest of this section presents two tools that can be used to perform exploration of mulithreaded Java programs as described earlier.

2.2.1 JPF

JPF[1] is a widely used framework for systematic exploration of Java programs [103]. JPF is essentially a JVM implementation (written using Java), which performs exploration of the multithreaded Java programs that it executes. It interprets the bytecode of a given Java program and maintains the dynamic state resulting from the execution of the program. When it encounters a bytecode that introduces non-determinism (e.g., schedule-relevant

[1]The name "JPF" stands for Java PathFinder.

events like field read/write and lock/unlock), it creates *choice points* containing the various possible choices (i.e., transitions) and explores them according to some defined exploration order. JPF stores and restores states to perform the exploration, and by default it performs stateful exploration. JPF provides a powerful API which allows users to customize all parts of the exploration including interpretation of bytecode, storing/restoring of states, and exploration/search order. JPF also provides a listener interface for observing the various events encountered during exploration. One of the implementations of the CAPP technique, which we describe in Chapter 4, was developed using the exploration customization API provided by JPF. Half of the experiments described in Chapter 4 were performed using this implementation. JPF is available publicly along with source code at: `http://babelfish.arc.nasa.gov/trac/jpf/`.

2.2.2 ReEx

ReEx[2] is a framework that we have built for systematic exploration of multithreaded Java code. ReEx uses dynamic bytecode instrumentation to add methods that can control the scheduling of threads. ReEx performs stateless exploration and uses re-execution to reestablish states during exploration. ReEx provides many useful facilities including:

- Deterministic replay of detected bugs.
- Listener API for observation of the various exploration events.
- Powerful API for customization of exploration.
- Close integration with JUnit.
- Support for `java.util.concurrent`.

One of the implementations of the CAPP technique, which we describe in Chapter 4, was developed using the exploration customization API provided by ReEx. We have used ReEx

[2]The name "ReEx" stands for Re-Execution based Exploration.

to perform the exploration of tests from many open-source programs. Half of the experiments described in Chapter 4 were performed using ReEx.

The ReEx framework consists of two main modules, instrumentation and exploration. The instrumentation module uses the ASM framework [83] to dynamically instrument bytecode to intercept all scheduling relevant instructions including field read/write, monitor enter/exit, synchronized method enter/exit, thread start/end/join/sleep/yield, object wait-/notify/notifyall, and park/unpark. The intercepted events are used to take control of the scheduling to enable exploration. The exploration module is used to make scheduling decisions which are enforced with the help of the instrumentation module. The exploration module exposes an API to allow for easy customization of the exploration. The main interfaces provided by the API include the `SchedulingStrategy` and `SchedulingFilter` interfaces. The exploration module also exposes a listener API to allow observation of the various exploration events through the `ExplorationListener` interface. ReEx has been released publicly along with source code at: `http://mir.cs.illinois.edu/reex`.

Chapter 3

IMUnit Framework

This chapter presents the contributions of the IMUnit framework that was developed with the aim of helping developers write improved multithreaded regression tests. This chapter is organized as follows. Section 3.1 presents our running example. Section 3.2 introduces the syntax and semantics of our language for expressing schedules. Section 3.3 describes our techniques for inferring events and schedules from legacy, sleep-based tests, and the tool for migrating legacy tests to IMUnit. Section 3.4 describes our tools for enforcing and checking execution of IMUnit schedules. Section 3.5 presents our evaluation of IMUnit.

3.1 Example

We now illustrate IMUnit with the help of an example multithreaded unit test for the ArrayBlockingQueue class in `java.util.concurrent` (JSR-166) [59]. ArrayBlockingQueue is an array-backed implementation of a bounded blocking queue. One operation 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 `add` and `take` operations execute on different threads. Consider testing some scenarios for these operations (in fact, the JSR-166 TCK provides over 100 tests for various scenarios for similar classes). Figure 3.1 presents an example multithreaded unit test for ArrayBlockingQueue

```
@Test
public void testTakeWithAdd() {
  ArrayBlockingQueue<Integer> q;
  q = new ArrayBlockingQueue<Integer>(1);
  new Thread(
    new CheckedRunnable() {
      public void realRun() {
        q.add(1);
        Thread.sleep(100);
        q.add(2);
      }
    }, "addThread").start();
  Thread.sleep(50);
  Integer taken = q.take();
  assertTrue(taken == 1 && q.isEmpty());
  taken = q.take();
  assertTrue(taken == 2 && q.isEmpty());
  addThread.join();
}
```

(a) JUnit

```
public class TestTakeWithAdd
    extends MultithreadedTest {
  ArrayBlockingQueue<Integer> q;
  @Override
  public void initialize() {
    q = new ArrayBlockingQueue<Integer>(1);
  }
  public void addThread() {
    q.add(1);
    waitForTick(2);
    q.add(2);
  }
  public void takeThread() {
    waitForTick(1);
    Integer taken = q.take();
    assertTrue(taken == 1 && q.isEmpty());
    taken = q.take();
    assertTick(2);
    assertTrue(taken == 2 && q.isEmpty());
  }
}
```

(b) MultithreadedTC

```
@Test
@Schedule("finishedAdd1->startingTake1,
           [startingTake2]->startingAdd2")
public void testTakeWithAdd() {
  ArrayBlockingQueue<Integer> q;
  q = new ArrayBlockingQueue<Integer>(1);
  new Thread(
    new CheckedRunnable() {
      public void realRun() {
        q.add(1);
        @Event("finishedAdd1")
        @Event("startingAdd2")
        q.add(2);
      }
    }, "addThread").start();
  @Event("startingTake1")
  Integer taken = q.take();
  assertTrue(taken == 1 && q.isEmpty());
  @Event("startingTake2")
  taken = q.take();
  assertTrue(taken == 2 && q.isEmpty());
  addThread.join();
}
```

(c) IMUnit

Figure 3.1: Example Multithreaded Unit Test for ArrayBlockingQueue

that exercises `add` and `take` in two 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 is likely to be 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 with and without blocking, when used with `add` from another thread. To check both scenarios, the test exercises schedules 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 13 shows the first sleep that is intended to pause the `main` thread[1] while the `addThread` finishes the first `add` operation. Line 9 shows the second sleep which is intended to pause the `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, there may be a false positive/negative. For example, if both `add` operations execute before a `take` is performed, the test will throw an exception and fail even if the code has no bug, and if both `take` operations finish without blocking, the test will not fail, even if the blocking `take` code had a bug.

Figure 3.1(b) shows the same test written using MultithreadedTC [88]. Note that it departs greatly from 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 a 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. The clock advances implicitly by one tick (or more ticks) when all threads are

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

blocked (and at least one thread is blocked in 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 code. It is especially not clear when `waitForTick` operations are blocked/unblocked, because ticks are advanced implicitly when all the threads are blocked.

Figure 3.1(c) shows the same test written using IMUnit. The interesting events encountered during test execution are marked with `@Event` annotations[2], 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 `addThread` should finish the first `add` operation before the `main` thread starts the first `take` operation, and that the `main` thread should block while performing the second `take` operation before the `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 schedules and code for both testing and debugging. In terms of *modularity*, note that IMUnit allows extracting the `addThread` as a helper thread (with its events) that can be reused in other tests (in fact, many tests in the JSR-166 TCK [59] use such helper threads). In contrast, reusing thread methods from the MultithreadedTC test class is more involved, requiring subclassing, parametrizing tick values, and/or providing appropriate parameter values. Also, IMUnit allows specifying multiple schedules for the same test code as discussed in Section 3.3.3. In terms of *relia-*

[2]Note that `@Event` annotations appear on statements. However, Java currently (version 7) does not support annotations on statements. For the time being, `@Event` annotations can be written as comments, e.g., `/* @Event("finishedAdd1") */`, which IMUnit Original translates into code for test execution. Alternatively, in IMUnit Light, events are specified through `fireEvent("eventName")` calls. Since `@Schedule` annotations appear on methods, they are already fully supported in the current version of Java.

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

Figure 3.2: Syntax of the IMUnit Schedule Language

bility, IMUnit does not rely on real time and hence has no false positives/negatives due to unintended schedules. In terms of *migration costs*, note that IMUnit tests resemble legacy JUnit tests more than MultithreadedTC tests. This similarity eases the transition of legacy tests into IMUnit: in brief, add `@Event` annotations, add `@Schedule` annotation, and remove `sleep` calls. Section 3.3 presents our techniques and tool that automate this transition by inferring relevant events and schedules.

3.2 Schedule Language

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

3.2.1 Concrete Syntax

Figure 3.2 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 qualified class name. There are two implicit event names for each thread, namely `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 3.1(c)). A *condition* is a conjunctive/disjunctive combination of basic events 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.

3.2.2 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 implemented in the future. The semantics of our logic is given in Section 3.2.3; 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 *block* operator and the *ordering* operator, respectively. For uniformity, all events are tagged with their thread. There are four implicit events: $\mathit{start}@t$ and $\mathit{end}@t$ were discussed above, and $\mathit{block}@t$ and $\mathit{unblock}@t$ correspond to when thread t gets blocked and unblocked[3].

For example, the following formula in our logic

$$
\begin{array}{rl}
 & (a_1@t_1 \wedge ([t_2] \vee (\neg(\mathit{start}(t_2) \to a_1@t_1)))) \to a_2@t_2 \\
\wedge & (a_2@t_2 \wedge ([t_1] \vee (\mathit{end}(t_1) \to a_2@t_2))) \to a_2@t_2
\end{array}
$$

says that if event a_2 is generated by thread t_2 then: (1) event a_1 must have been generated

[3]It is expensive to explicitly generate *block*/*unblock* events in Java precisely when they occur, because it requires polling the status of each thread; our currently implemented fragment only needs, through its restricted syntax, to check if a given thread is currently blocked or not, which is fast.

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 present the precise semantics, we explain how our current IMUnit language shown in Figure 3.2 (whose design was driven exclusively by practical needs) is a smaller 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 in practice, IMUnit currently disallows their explicit use in schedules. Moreover, to reduce their implicit use to a simple and 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 $((\varphi \wedge a_1@t_1) \rightarrow a_2@t_2) \wedge ((a_2@t_2 \wedge [t_1]) \rightarrow a_2@t_2)$, relying on $a_2@t_2$ happening at most once.

3.2.3 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 $\tau = e_1 e_2 \ldots e_n$. We therefore assume that only one event can take place at a time. Even though in principle one can rightfully argue that events may take place concurrently in a multithreaded system, we observe and enforce program executions by means of monitors, which process one event at a time. It is only these monitors

that need to be aware of the semantics of schedules. 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 $a \notin \{block, unblock\}$.

The semantics of our logic is defined as follows:

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

$\tau \vDash \varphi \ \vee \ \psi$ iff $\tau \vDash \varphi$ or $\tau \vDash \psi$

$\tau \vDash \varphi \ \wedge \ \psi$ iff $\tau \vDash \varphi$ and $\tau \vDash \psi$

$e_1 e_2 \ldots e_n \vDash [t]$ iff $(\exists 1 \leq i \leq n)$ (e_i = *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 $(\exists 1 \leq j < i)$ $e_1 e_2 ... e_j \vDash \varphi$)

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

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

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

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

3.3 Migration

We now describe the process of migrating legacy, sleep-based tests to IMUnit, event-based tests. First we present the steps that are typically performed during manual migration and then we describe the automated support that we have developed for key steps of the migration.

3.3.1 Manual Migration

Based on our experience of manually migrating over 200 tests, the migration process typically follows these steps:

Step 1: Optionally add explicit names for threads in the test code (by using a thread constructor with a name or by adding a call to `setName`). This step is required if events are tagged with their thread name (e.g. `finishedAdd1@addThread`) in the schedule, because by default the JVM automatically assigns a name (e.g. `Thread-5`) for each thread created without an explicit name, and the automatic name may differ between JVMs or between different runs on the same JVM.

Step 2: Introduce `@Event` annotations for the events relevant for the intended schedule. Some of these annotations will be used for block events and some for basic events.

Step 3: Introduce a `@Schedule` annotation for the intended schedule. Steps 2 and 3 are the hardest to perform as they require understanding the intended behavior of the sleep-based test. Note that a schedule with too few orderings can lead to failing tests that are false positives. On the other hand, a schedule with too many orderings may lead to false negatives whereby a bug is missed because the schedule is over-constraining the test execution.

Step 4: Check that the orderings in the introduced schedule are actually satisfied when running the test with sleeps (Section 3.4 describes the passive, checking mode).

```
enum EntryType { SLEEP_CALL, SLEEP_RETURN, BLOCK_CALL, BLOCK_RETURN,
                 OTHER_CALL, OTHER_RETURN, TH_START, TH_END, EVENT }
class LogEntry { EntryType type; ThreadID tid; String info; StmtID sid; }
```

Figure 3.3: Log Entries

Step 5: Remove sleeps.

Step 6: Optionally merge multiple tests with different schedules (but similar test code) into one test with multiple schedules, potentially adding schedule-specific code and assertions (Section 3.3.3 discusses this).

3.3.2 Automated Migration

We have developed automated tool support to enable easier migration of sleep-based tests to IMUnit. In particular, we have developed inference techniques that can compute likely relevant events and schedules for sleep-based tests by inspecting the execution logs obtained from test runs. We next describe the common infrastructure for logging the test runs. We then present the techniques for inferring events and schedules.

Lightweight Logging

Our inference of events and schedules from sleep-based tests is dynamic: it first instruments the test code (using AspectJ [66]) to emit entries potentially relevant for inference, then runs the instrumented code (several times, as explained below) to collect logs of entries from the test executions, and finally analyzes the logs to perform the inference.

Figure 3.3 shows the generic representation for log entries, although event and schedule inferences require slightly different representations. Each log entry has a type, name/ID of the thread that emits the entry, potential info/parameters for the entry, and the ID of the statement that creates the entry (which is used only for event inference). The types of log entries and their corresponding `info` are as follows:

- `SLEEP_CALL`: Invocation of `Thread.sleep` method. (Only used for inferring events.)
- `SLEEP_RETURN`: Return from `Thread.sleep` method.
- `BLOCK_CALL`: Invocation of a thread blocking method (`LockSupport.park` or `Object.wait`).
- `BLOCK_RETURN`: Return from a thread blocking method.
- `OTHER_CALL`: Invocation of a method (other than those listed above) in the test class. The `info` is the method name. (Only used for inferring events.)
- `OTHER_RETURN`: Return from a method executed from the test class. This type includes log entries for the `run` methods that start thread execution.
- `TH_START`: Invocation of `Thread.start`. The `tid`, as usual, has the ID of the thread calling `start`, and the `info` is the ID of the started thread. (Only used for inferring schedules.)
- `TH_END`: End of thread execution.
- `EVENT`: Execution of an IMUnit event. The `info` is the name of the event. (Only available while inferring schedules.)

Note that any logging can affect timing of test execution. Because sleep-based tests are especially sensitive to timing, care must be taken to avoid false positives. We address this in three ways. First, our logging is lightweight. The instrumented code only collects log entries (and their parameters) relevant to the inference. For example, `OTHER_CALL` is not collected for schedule inference. Also, the entries are buffered in memory during test execution, and they are converted to strings and logged to file only at the end of test execution. While keeping entries in memory would not work well for very long logs, it works quite well for the relatively short logs produced by test executions. Second, our instrumentation automatically scales the duration of sleeps by a given constant N to compensate for the logging overhead. For

example, for $N = 3$ it increases all sleep times 3x. Increasing all the durations almost never makes a passing test fail, but it does make the test run slower. Third, we perform multiple runs of each test and only collect logs for passing runs. This increases the confidence that the logs indeed correspond to the intended schedules specified with sleeps.

Inferring Events

Figure 3.4 presents the algorithm for inferring IMUnit events from a sleep-based test. The input to the algorithm consists of a set of logs (as described in Section 3.3.2) and a `confidenceThreshold` that determines what percentage of logs need to produce the same result before that result is used for the entire inference. The output is a set of inferred events. Each event includes the code location where `@Event` annotation should be added and the name of the event. The intuition behind the algorithm is that `SLEEP_CALL` log entries are indicative of code locations for events. More precisely, a thread t calls `sleep` to wait for one or more events to happen on other threads (those will be "finished" events) before an event happens on t (that will be a "starting" event). Recall our example from Section 3.1. When the `main` thread calls `sleep`, it waits for `add` to finish before `take` starts, and thus `finishedAdd1` executes before `startingTake1`.

For each log, the algorithm first computes a set of *regions*, each of which is a sequence of log entries between `SLEEP_CALL` and the matching `SLEEP_RETURN` executed by the same thread. The log entries executed by other threads within a region are potential points for the "finished" events. Regions from different threads can be partially or completely overlapping, but regions from the same thread are disjoint. Figure 3.5 shows two regions that are formed for a simplified log produced by our running example. In pseudo-code, each region is represented as a pair of `int`s that point to the beginning and end of the region in the list of log entries. For each region, the algorithm first calls `addFinishedEvents` to potentially add some "finished" events for threads other than the region's thread. If such an event is added, the algorithm calls `addStartingEvent` to add the matching "starting" event.

```
1  class StaticEvent { StmtID sid; String name; }
2  class Region { int start; int end; }
3  // Input
4  Set⟨List⟨LogEntry⟩⟩ logs; float confidenceThreshold;
5  // Output
6  Set⟨StaticEvent⟩ events;
7  // State
8  Bag⟨StaticEvent⟩ inferred := ∅;
9
10 void inferEvents() {
11   foreach (List⟨LogEntry⟩ log in logs) {
12     foreach (Region r in computeRegions(log)) {
13       boolean addedFinished := addFinishedEvents(r, log);
14       if (addedFinished) { addStartingEvent(r, log); }
15     }
16   }
17   filterOutLowConfidence(confidenceThreshold); events := inferred.toSet();
18 }
19 Set⟨Region⟩ computeRegions(List⟨LogEntry⟩ log) {
20   return { new Region(i, j) | log(i).type = SLEEP_CALL ∧
21       j := min{ k | log(i).tid = log(k).tid ∧ log(k).type = SLEEP_RETURN } }
22 }
23 boolean addFinishedEvents(Region r, List⟨LogEntry⟩ log) {
24   boolean result := false;
25   foreach (ThreadID t in { log(i).tid | i ∈ r } − { log(r.start).tid }) {
26     Set⟨int⟩ relevant := { i ∈ r | log(i).tid = t ∧ log(i).type ∈ { SLEEP_CALL, BLOCK_CALL, TH_END } ∧
27           ¬(∃ j ∈ r | log(j).tid = t ∧ log(j).type ∈ { SLEEP_RETURN, BLOCK_RETURN }) }
28     if (relevant.size() ≠ 1) continue;
29     int starting := max{ j < relevant | log(j).tid = t ∧ log(j).type = OTHER_RETURN }
30     addEvent(relevant, "finished", starting); result := true;
31   }
32   return result;
33 }
34 void addStartingEvent(Region r, List⟨LogEntry⟩ log) {
35   int finished := min{ j > r.start | log(j).tid = log(r.start).tid ∧ log(j).type ∈ { OTHER_CALL, TH_END }}
36   addEvent(finished, "starting", finished);
37 }
38 void addEvent(int location, String namePrefix, int suffixIdx) {
39     StmtID sid = log(location).sid;
40     events ∪= new StaticEvent(sid, namePrefix + log(suffixIdx).info + sid);
41 }
```

Figure 3.4: Events-Inference Algorithm

The procedure `addFinishedEvents` potentially adds an inferred event for each thread that executes at least one statement in the region. For each such thread, the procedure first discovers a *relevant* statement, which is one of `SLEEP_CALL`, `BLOCK_CALL`, and `TH_END`. Only threads that have exactly one relevant statement in the region are considered. The intuition is that sleeps usually wait for exactly one event in each other thread. If a thread executes none or multiple relevant statements, it is most likely independent of the thread that

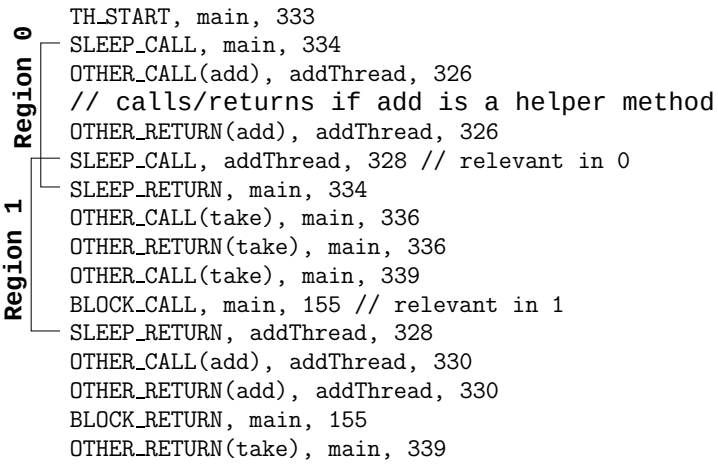

Figure 3.5: Snippet from a Log for Inferring Events

started the region and therefore can be ignored. Figure 3.5 shows the relevant statements for each region. The procedure then finds the `OTHER_RETURN` statement immediately before the relevant statement for each thread. This statement determines the name for the new "finished" `StaticEvent`, whereas the relevant statement determines the location. Note that logging only method calls would not be enough to properly determine the previous statement since the call can come from a helper method in the test class. For our example, these before log entries are `OTHER_RETURN(add), addThread, 326` and `OTHER_RETURN(take), main, 336` (Figure 3.5).

The procedure `addStartingEvent` adds an event for the thread that starts the region. The event is placed just before the first statement that follows the end of the region. The type of the statement can be any, including `OTHER_CALL`. The same statement is used for naming the event. In Figure 3.5, the algorithm finds `OTHER_CALL(take), main, 336` and `OTHER_CALL(add), addThread, 330`.

Inferring Schedules

Figure 3.6 presents the algorithm for inferring an IMUnit schedule for a sleep-based multi-threaded unit test that already contains IMUnit event annotations. These annotations can be automatically produced by our event inference or manually provided by the user. The input

to the algorithm is a set of logs obtained from the passing executions of the sleep-based test. Figure 3.7 shows a snippet from one such log for our running example sleep-based test shown in Figure 3.1(a). The input also contains a `confidenceThreshold` which will be described later. The output is an inferred schedule, i.e., a set of orderings that encodes the intended schedule for the test. The main part of the algorithm is the `addSleepInducedOrderings` procedure. It captures the intuition that a thread normally executes a sleep to wait for the other active threads to perform events. Recall line 13 from our example in Figure 3.1(a) where the `main` thread sleeps to wait for the thread `addThread` to perform an `add` operation, and line 9 where the thread `addThread` sleeps to wait for the `main` thread to first perform one `take` operation and then block while performing the second `take` operation.

For each log, the procedure scans for `SLEEP_RETURN` entries (line 29). As shown in Figure 3.7, the log for our example contains two `SLEEP_RETURN` entries, one each in the `main` thread and `addThread`. For each `SLEEP_RETURN` found, the procedure does the following:

1. Retrieves the next `EVENT` entry for the same thread (line 30). This event will be used as the `after` event in the orderings induced by the `SLEEP_RETURN`. In the example log, the two `after` events are `startingTake1` for the first `SLEEP_RETURN` and `startingAdd2` for the second `SLEEP_RETURN`.

2. Computes the other threads that were *active* between the `SLEEP_RETURN` and the `after` event (line 31). In the example, for the first `SLEEP_RETURN`, the only other active thread is `addThread` and for the second `SLEEP_RETURN`, the only other active thread is the `main` thread.

3. Finds for each active thread the last `EVENT` entry in the log that is before the `after` event. This event will be used as the `before` event in the `Ordering` induced by the `SLEEP_RETURN` with the corresponding active thread (line 34). It is important to note that this `before` event on another thread can be even *before* the `SLEEP_RETURN`. Effectively, this event is the *current* last entry and not the last entry at the time of the

```
1  class Event { String eventName; ThreadID tid; }
2  abstract class Ordering { Event before; Event after; }
3  class NonBlockingOrdering extends Ordering {}
4  class BlockingOrdering extends Ordering {}
5  // Input
6  Set⟨List⟨LogEntry⟩⟩ logs; float confidenceThreshold;
7  // Output
8  Set⟨Ordering⟩ orderings;
9  // State
10 Bag⟨Ordering⟩ inferred := ∅;
11
12 void inferSchedules() {
13   foreach (List⟨LogEntry⟩ log in logs) {
14     List⟨LogEntry⟩ preprocessed := preprocessLog(log); addSleepInducedOrderings(preprocessed);
15   }
16   minimize();
17 }
18 List⟨LogEntry⟩ preprocessLog(List⟨LogEntry⟩ log) {
19   List⟨LogEntry⟩ result := log.clone();
20   foreach ({ i | log(i).type = SLEEP_RETURN }) {
21     int j := min{j > i | log(j).tid = log(i).tid };
22     if (log(j).type = TH_START) {
23       result(j) := new LogEntry(SLEEP_RETURN, _, log(j).info); result(i) := log(j);
24     }
25   }
26   return result;
27 }
28 void addSleepInducedOrderings(List⟨LogEntry⟩ log) {
29   foreach ({ i ∈ log.indexes() | log(i).type = SLEEP_RETURN }) {
30     ThreadID t := log(i).tid; int j := min{ n > i | log(n).tid = t ∧ log(n).type = EVENT };
31     Set⟨ThreadID⟩ active := { t' | ( ∃ n < j | log(n).tid = t' ∧ log(n).type = EVENT ) ∧
32                                  ( ∃ n > i | log(n).tid = t' ∧ log(n).type = TH_END ) };
33     foreach (ThreadID t' in active − { t }) {
34       int j' := max{ n < j | log(n).tid = t' ∧ log(n).type = EVENT };
35       Event before := new Event(log(j').info, t'); Event after := new Event(log(j).info, t);
36       if (log(min{ n > j' | log(n).tid = t' }).type ≠ BLOCK_CALL) {
37         inferred ∪= new NonblockingOrdering(before, after);
38       } else { // before.type = BLOCK_CALL
39         inferred ∪= new BlockingOrdering(before, after);
40       }
41     }
42   }
43 }
44 void minimize(List⟨LogEntry⟩ log) {
45   Set⟨Ordering⟩ graph := inferred.toSet() ∪ computeSeqOrderings(log);
46   removeCyclicOrderings(graph); performTransitiveReduction(graph);
47   inferred.onlyRetainOrderingsIn(graph); filterOutLowConfidence(confidenceThreshold);
48   orderings := inferred.toSet();
49 }
50 void Set⟨Ordering⟩ computeSeqOrderings(List⟨LogEntry⟩ log) {
51   return { new NonblockingOrdering(log(i), log(j)) | i < j ∧ log(i).tid = log(j).tid ∧
52                            log(i).type = log(j).type = EVENT ∧ ¬(∃ k | i < k < j ∧
53                            log(j).tid = log(k).tid ∧ log(k).type = EVENT) };
54 }
```

Figure 3.6: Schedule-Inference Algorithm

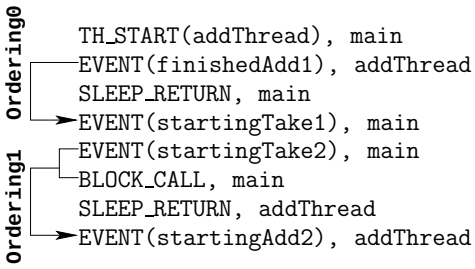

Figure 3.7: Snippet from a Log for Inferring Schedules

sleep. In the example, the two `before` events are `finishedAdd1` and `startingTake2` for the first and second `SLEEP_RETURN` events, respectively.

4. Creates an `Ordering` for each `before` and `after` event pair and inserts it into the `inferred` bag. If a `before` event is followed immediately by a `BLOCK_CALL` (within entries for the same thread), a `BlockingOrdering` is created; otherwise, a `NonBlockingOrdering` is created (line 36). In the example, since `startingTake2` is followed by a `BLOCK_CALL`, the ordering between `startingTake2` and `startingAdd2` will be a `BlockingOrdering`, while the other ordering between `finishedAdd1` and `startingTake1` will be a `NonBlockingOrdering`.

Before the `addSleepInducedOrderings` procedure is invoked, each `log` is modified by the `preprocessLogs` procedure. This procedure looks for `SLEEP_RETURN` entries followed immediately by `TH_START` entries for the same thread. For every such instance, it swaps the `SLEEP_RETURN` and `TH_START` entries and sets the `tid` of the `SLEEP_RETURN` entry to be the ID of the thread that is *started* by the `TH_START` event. The intuition is that a `SLEEP_RETURN` followed by a `TH_START` signifies that the *started* thread, rather than the *starting* thread performing the `TH_START`, should wait for the other active threads to perform events. Many of the sleep-based tests that we migrated included instances of this pattern. Effectively, this swap makes it appear as if the sleep was at the beginning of the `run` method for the started thread, although the sleep was actually before the `start` method.

After each log is processed by the `preprocessLogs` and `addSleepInducedOrderings` proce-

dures, the `inferred` bag is populated with all the inferred orderings. However, the inferred orderings may contain cycles (e.g., `a->b` and `b->a`) and transitively redundant orderings (e.g., `a->b`, `b->c`, and `a->c`, where the last ordering is redundant). The `minimize` procedure removes such orderings. It first creates an ordering `graph` by combining the edges from the `inferred` orderings with the edges implied by the sequential orderings of events within each thread (the latter edges being computed by the `computeSeqOrderings` procedure). It then removes all the edges of the `graph` that participate in cycles. It finally performs a transitive reduction on the acyclic `graph` and updates the `inferred` bag by removing all orderings not included in the reduced `graph`. We use an open-source implementation [30] of the transitive reduction algorithm introduced by Aho et al. [3]. Since the transitive reduction is performed on an acyclic `graph`, we can use a specialized case of the general algorithm.

The last step of the `minimize` procedure is to remove the orderings that were inferred with low confidence. Recall that the input to our inference is a set of logs from several (passing) runs of the test being migrated. The confidence of an inferred ordering is the ratio of the count of that ordering in the `inferred` bag and the number of logs/runs. For example, an ordering may be inferred in only 60% of runs, say 3 out of 5. The `confidenceThreshold` defines the lowest acceptable confidence. All inferred orderings with confidence lower than the specified threshold are discarded. In Section 3.5 we discuss the impact of the various steps of `minimize` on the inference of schedules for the sleep-based tests that we migrated to event based IMUnit tests.

Eclipse Plugin

We have developed a refactoring plugin for the popular Eclipse [44] IDE to enable automated migration of existing sleep-based unit tests into event-based IMUnit tests. The plugin is implemented using the generic refactoring API provided by the Eclipse `LTK` (Language Toolkit) [45]. The refactoring automates the most important steps required to migrate an existing sleep-based unit test into an IMUnit test: the introduction of events and schedule

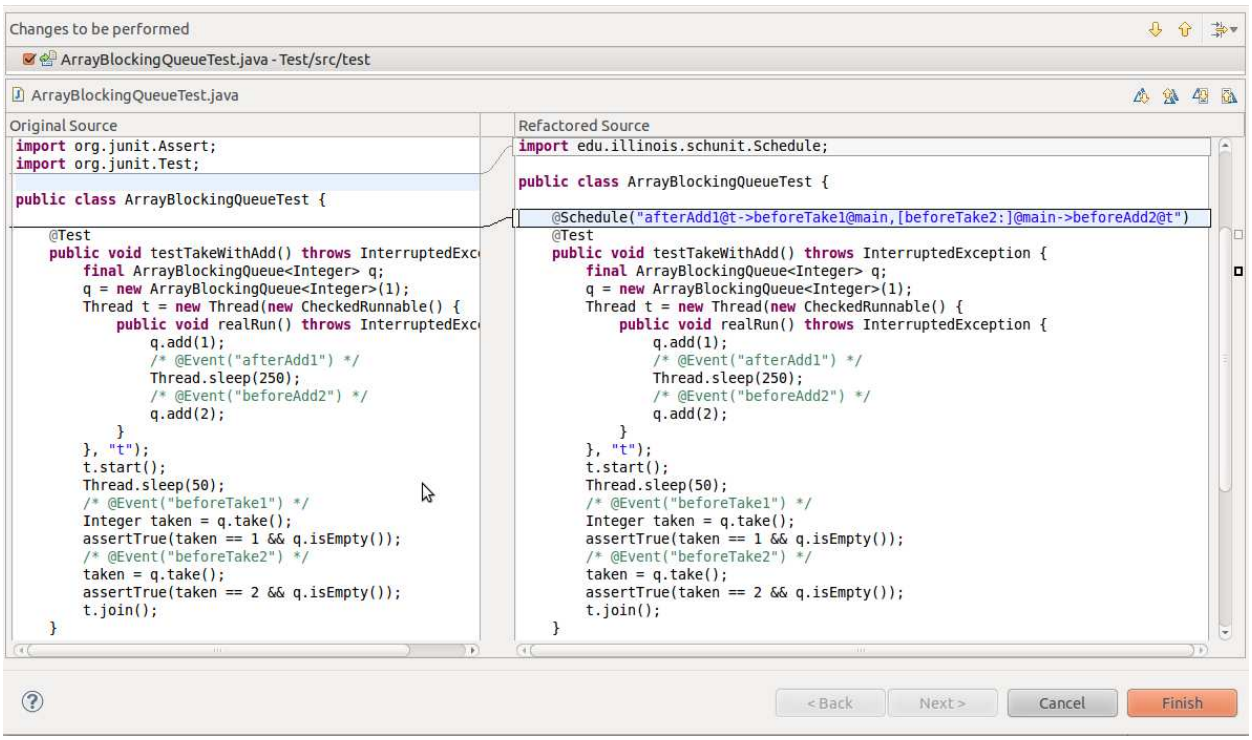

Figure 3.8: Application of Refactoring on Running Example

(using inference techniques) and checking of the introduced schedule. The refactoring also provides some support to optionally help the user name the threads used in the test.

The refactoring is applied on a sleep-based test and performs the following actions:

1. Finds the thread instances that are initialized by the test and warns the user if a thread name parameter has not been provided since the thread could then be named differently across runs.

2. Optionally extracts the unnamed thread instances into local variables (if not already assigned to one) to allow the user to set the thread name and join the thread at the end of the test.

3. Instruments and executes the appropriate code and libraries to the collect passing logs with the log entries required for event inference.

4. Performs event inference using the passing logs and inserts the inferred events as `@Event`

annotations into the code for the sleep-based test.

5. Instruments and executes the test code with the inferred events to collect passing logs with the log entries required for schedule inference.

6. Performs schedule inference with the passing logs and inserts the inferred schedules as `@Schedule` annotations at the beginning of the sleep-based test method.

7. Presents a preview of the transformations for the developers approval.

Figure 3.8 shows a screen shot demonstrating the application of the refactoring on the example described in Section 3.1.

3.3.3 Multiple Schedules

As mentioned in Step 6 of Section 3.3.1, after converting sleep-based tests to event-based IMUnit tests, developers can merge several similar tests with different schedules into one test with multiple IMUnit schedules. Recall our example sleep-based test from Figure 3.1(a). Its intended schedule is an `add` followed by a non-blocking `take` and a blocking `take` followed by another `add`. Suppose that the same test class contained another sleep-based test whose indented schedule is an `add` followed by a non-blocking `take` and another `add` followed by another non-blocking `take`. Although these two sleep-based tests would be almost identical (with the sleep at line 9 moved to before line 16), they cannot share the common code without using additional conditional statements to enable the appropriate sleeps during execution. In contrast, after both tests are migrated to event-based IMUnit tests, they can be easily replaced by just one new test as shown in Figure 3.9. This new test has the same code as in Figure 3.1(a), with two added annotations: (1) `@Event("finishedAdd2")` added after the `add(2)` call, and (2) `@Schedule("finishedAdd1->startingTake1, finishedAdd2->startingTake2")` added before the test method. Note that Java does not allow multiple instances of an annotation for a method. As a workaround, IMUnit provides the `@Schedules` annotation which

```
1  @Test
2  @Schedules({
3  @Schedule("finishedAdd1->startingTake1,
4              [startingTake2]->startingAdd2"),
5  @Schedule("finishedAdd1->startingTake1,
6              finishedAdd2->startingTake2") })
7  public void testTakeWithAdd() {
8    ArrayBlockingQueue<Integer> q;
9    q = new ArrayBlockingQueue<Integer>(1);
10   new Thread(
11     new CheckedRunnable() {
12       public void realRun() {
13         q.add(1);
14         @Event("finishedAdd1")
15         @Event("startingAdd2")
16         q.add(2);
17         @Event("finishedAdd2")
18       }
19     }, "addThread").start();
20   @Event("startingTake1")
21   Integer taken = q.take();
22   assertTrue(taken == 1 && q.isEmpty());
23   @Event("startingTake2")
24   taken = q.take();
25   assertTrue(taken == 2 && q.isEmpty());
26   addThread.join();
27 }
```

Figure 3.9: Example IMUnit Test with Multiple Schedules

takes an array of `@Schedule` annotations as its parameter, and thus allows specification of multiple schedules for a test method.

In this discussion, the two tests had exactly the same code and only different schedules. In general, the code for the two tests can be slightly different. IMUnit supports that case with two constructs. First, IMUnit allows the schedule annotations to optionally provide a name for each schedule. Second, IMUnit provides a method `IMUnit.getCurrentSchedule` that returns the name of the schedule being enforced while running the test with the IMUnit execution tool.

3.4 Enforcement and Checking

We now describe our tools for enforcing/checking the schedules specified in IMUnit multithreaded unit tests. Our first tool, called IMUnit Original, implements all parts of the

IMUnit schedule language but requires more effort to set up and use because of additional pre-processing and dependence on JavaMOP and AspectJ. The second tool, called IMUnit Light, was developed to ease the adoption of IMUnit and is distributed and used as a single Java jar file, but it does not implement all language features. We first present the details of IMUnit Original and then present how IMUnit Light differs from it.

3.4.1 IMUnit Original

This was the first implementation of IMUnit. It consists of three main parts:

1. A pre-processor that converts `/* @Event("eventName") */` annotations into appropriate framework method calls.

2. A monitor generator that processes `@Schedule` annotations and generates one monitor (in the form of an aspect) for each schedule and test pair. Each generated monitor has two operation modes. In the *active mode*, it controls the thread scheduler to enforce an execution of the test to satisfy the given schedule. Note that this mode avoids the main problem of sleep-based tests, that of false positives and negatives due to the execution of unintended schedules. In the *passive mode*, our tool observes and checks the execution provided by the JVM against the given schedule. The passive mode is particularly useful for checking whether executions enforced by the tool for some schedules (e.g., automatically inferred ones as in Section 3.3) satisfy other schedules (e.g., the manually provided ones).

3. A custom JUnit runner that executes each test once with each of the corresponding generated monitors weaved in. The runner also performs automated deadlock detection to detect general deadlocks as well as deadlocks caused by incorrect schedules.

A key part of this implementation is monitor generation which is implemented using JavaMOP [25, 77], 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 3.2), 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 3.2. This plugin synthesizes monitors that either enforce or check a given IMUnit schedule, depending on the running 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. The plugin takes as input an IMUnit schedule and test pair and generates as output a monitor written in pseudo-code; a *Java shell* for this language then turns the monitor into AspectJ code [66], 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 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 3.2). 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.

As an example, Figure 3.10 shows the active-mode monitor generated for the schedule in Figure 3.1(c). When events `finishedAdd1` and `startingTake2` occur, the monitor just sets the corresponding Boolean variables, as there is no condition for those events. For event

```
switch (event) {
  case finishedAdd1:
    occurred_finishedAdd1 = true;
    notifyAll();
  case startingTake2:
    thread_startingTake2 = currentThread();
    occurred_startingTake2 = true;
    notifyAll();
  case startingTake1:
    while (!occurred_finishedAdd1) {
      wait();
    }
    occurred_startingTake1 = true;
    notifyAll();
  case startingAdd2:
    while (!(occurred_startingTake2 && isBlocked(thread_startingTake2))) {
      wait();
    }
    occurred_startingAdd2 = true;
    notifyAll();
}
```

Figure 3.10: Monitor for the Schedule in Figure 3.1(c)

`startingTake1`, it checks if there was an event `finishedAdd1` in the past by checking the variable `occurred_finishedAdd1`; if not, the thread will be blocked until `finishedAdd1` occurs. For event `startingAdd2`, in addition to checking the Boolean variable for `startingTake2`, it also checks whether the thread of the event `startingTake2` is blocked; if not, the thread of the event `startingAdd2` will be blocked until both conditions are satisfied. Note that the user may have specified an infeasible schedule, which can cause a deadlock where all threads are paused waiting for infeasible events. Our custom JUnit runner includes a low-overhead runtime deadlock detection mechanism that detects and reports such deadlocks (and other general deadlocks). This is achieved by executing each test within a separate thread group and monitoring the status of all the threads in each test thread group.

3.4.2 IMUnit Light

We developed IMUnit Light specifically to ease the adoption and use of IMUnit. IMUnit Light does not require any pre-processing and is not dependent on JavaMOP or AspectJ. It is distributed and used as a single Java jar. IMUnit Light has been released at

`http://mir.cs.illinois.edu/imunit` and has already garnered some interest, including developers of the Apache River project using it to write their multithreaded unit tests. The main differences between IMUnit Original and IMUnit Light are as follows:

1. Use of `fireEvent("eventName")` method calls instead of `/* @Event("eventName") */` annotations to specify events.

2. The `fireEvent("eventName")` method calls invoke a central implementation of a generic monitor that can enforce/check any single schedule. The schedule to be enforced/checked is set at the beginning of a test and unset at the end of a test. Hence, there is no need for generation of monitors.

3. The generic monitor implementation does not support Boolean combination of events.

IMUnit Light also contains a custom JUnit runner that executes the tests, and excluding these changes, IMUnit Light supports all the features supported by IMUnit Original including automated deadlock detection. Note that IMUnit Light demonstrates considerable performance benefits compared to IMUnit Original, but the results presented in Section 3.5 are for IMUnit Original.

3.5 Evaluation

To evaluate the IMUnit contributions—schedule language, automated migration, and schedule execution—we analyzed over 200 sleep-based tests from several open-source projects. Table 3.1 lists the projects and the number of sleep-based tests that we manually migrated to IMUnit. We first describe our experience with the IMUnit language. We then present quantitative results of our inference techniques for migration. We finally discuss the test running time with IMUnit execution. Note that the evaluation was performed with the IMUnit Original implementation.

3.5.1 Schedule Language

It is hard to quantitatively evaluate and compare languages, be it implementation or specification languages, including languages for specifying schedules. One metric we use is how *expressive* the language is, i.e., how many schedules from sleep-based tests can be expressed in IMUnit such that *sleeps can be removed altogether*. Note that IMUnit conceptually subsumes sleeps: sleeps and IMUnit events/schedules can co-exist in the same test, and developers just need to make sleeps long enough to account for the IMUnit schedule execution/enforcement. While every sleep-based test is trivially an IMUnit test, we are interested only in those tests where IMUnit allows removing sleeps altogether.

We were able to remove sleeps from 198 tests, in fact all sleeps from all but 4 tests. While the current version of IMUnit is highly expressive, we have to point out that we refined the IMUnit language based on the experience with migrating the sleep-based tests. When we encountered a case that could not be expressed in IMUnit, we considered how frequent the case is, and how much IMUnit would need to change to support it. For example, blocking events are very frequent, and supporting them required a minimal syntactic extension (adding events with square brackets) to the initial version of our language. However, some cases would require bigger changes but are not frequent enough to justify them. The primary example is events in a loop. IMUnit currently does not support the occurrence of an event more than once in a trace. We did find 4 tests that would require multiple event occurrences, but changing the language to support them (e.g., adding event counters or loop indices to events) would add a layer of complexity (to the language syntax/semantics and the time for schedule execution) that is not justified by the small number of cases. However, as we apply IMUnit to more projects, and gain more experience, we expect that the language could grow in the future.

Subject	Source	Tests	Events	Orderings
Collections	[6]	18	51	32
JBoss-Cache	[60]	27	105	47
Lucene	[8]	2	3	4
Mina	[9]	1	2	1
Pool	[7]	2	8	3
Sysunit	[27]	9	33	34
JSR-166 TCK	[59]	139	577	277
$\sum$		198	779	398

Table 3.1: Subject Programs Statistics

3.5.2 Inference of Events and Schedules

To measure the effectiveness of our migration tool in inferring events/schedules, we calculated precision and recall of automatically inferred events/schedules with respect to the manually written events/schedules (i.e., the manual translations from sleep-based schedules). We used the standard definitions of precision and recall from the information retrieval community [75]. Precision was defined as the percentage of inferred events/schedules that matched manually written events/schedules. Recall was defined as the percentage of manually written events/schedules for which matching events/schedules were inferred. So, in order to compute precision and recall, we had to compare the automatically inferred events/schedules with the manually written events/schedules. For event inference, the input is a sleep-based test, and the output is a set of events. Our current comparison uses only the *source-code location* (line number) of the static events and not their name, but it requires the exact match for locations. For schedule inference, the input is a sleep-based test *with manually written (not automatically inferred) events*, and the output is a schedule. Our comparison considers all orderings from the automatically inferred and manually written schedules; it considers that two orderings match only if they have exactly the same *both before and after events* (including their name, optional thread ID, and type that can be basic or blocking). We performed the comparisons for all but 14 (discussed below) of our 198 tests. Table 3.2 shows for each project precision and recall values, averaged over the tests from that project.

Subject	Inferring Events		Inferring Schedules	
	Precision	Recall	Precision	Recall
Collections	0.75	0.82	0.96	0.97
JBoss-Cache	0.83	0.86	0.87	0.96
Lucene	0.75	1.00	1.00	0.75
Mina	0.22	1.00	1.00	1.00
Pool	0.90	1.00	1.00	1.00
Sysunit	0.76	0.87	0.89	0.89
JSR-166 TCK	0.67	0.74	0.98	0.98
Overall	0.75	0.79	0.96	0.94

Table 3.2: Precision and Recall for Inference

Columns two and three from Table 3.2 show the results for event inference. In most cases, precision and recall are fairly high. We inspected the cases with lower precision and identified two causes for it. The first cause is due to our evaluation setup and not the algorithm itself. Namely, our current comparison requires the exact match of source-code locations. If the locations differ, the inferred event counts as a false negative, even if it was only a few lines from the manually written event, and even if those locations are equivalent with respect to the code. In the future, one could improve the setup by analyzing the code around the automatically inferred and manually written events to determine if their locations are equivalent. The second reason is that some tests use sleeps that are not relevant for the thread schedule (e.g., JBoss-Cache has such sleeps in the helper threads shared among tests, and Lucene has similar sleeps while interacting with the I/O library). These extra sleeps mislead our inference, which assumes that every sleep is relevant for the schedule and infers events for every sleep.

Columns four and five from Table 3.2 show the results for schedule inference. The results are even more impressive than for event inference, with precision and recall of over 75% in all cases. We identified two causes for misses. The first cause is that some threads can be independent. The algorithm always forms edges from all threads to the thread that invokes sleep method, but this should not be done for independent threads. In the future, one could

consider an abstraction similar to regions (Figure 3.4) as a mechanism to detect independent threads. The second cause is the same as for event inference, namely unnecessary sleeps.

A known issue in information retrieval is that some result sets may be empty, which effectively corresponds to infinite precision and zero recall. For 14 of 198 tests, our inference techniques returned empty sets of events/schedules because these tests do not use sleeps to control schedules. Instead, these tests use `while (condition) { Thread.sleep/yield }`, `wait/notify`, `AtomicBoolean`, `CountDownLatch`, or other concurrent constructs to control schedules. We excluded these 14 tests from the evaluation of our inference techniques.

Our inference algorithms use `confidenceThreshold` to select some of the events/schedules, with the default value of 0.5 (for Table 3.2). We performed a set of experiments to evaluate how sensitive our inference is to the value of `confidenceThreshold`. We found that the results are quite stable. For example, for schedule inference, when changing the value from 0.5 to 0.1, only for Lucene the precision drops from 1 to 0.75 (while is stays the same for all other programs). When changing the value from 0.5 to 0.9, only for JBoss-Cache the precision and recall drop from 0.87 and 0.96 to 0.86 and 0.93, respectively. For all other cases, everything else is inferred exactly the same for the values 0.1 and 0.9 as for the default value 0.5.

The other input to our inference algorithms is the set of logs obtained from passing runs of the legacy tests. By default, we collect 5 passing logs for each test (for Table 3.2). Different runs of the legacy test can produce different logs that can in turn result in different sets of events/schedules being inferred. Therefore, depending on the number of logs, inferred events/schedules could differ. So we evaluated how sensitive our inference is to the number of logs. We found that the logs are quite stable, and almost identical results were obtained for 1, 5, and 10 logs. For instance, going from 5 to 10 logs only the recall for JBoss-Cache drops from 0.96 to 0.94, and everything else remains the same.

Lastly, our schedule-inference algorithm runs a minimization phase after processing all the logs. Table 3.3 summarizes the results of this phase. It tabulates, for each project, the number of schedule orderings originally inferred before minimization (Original) and the

Subject	Original	CR	TR	LC
Collections	33	0	0	0
JBoss-Cache	39	2	3	0
Lucene	5	0	1	1
Mina	1	0	0	0
Pool	3	0	0	0
Sysunit	39	0	5	0
JSR-166 TCK	306	0	30	1
$\sum$	426	2	39	2

Table 3.3: Numbers of Removed Orderings

Subject	Original [s]	IMUnit [s]		Speedup	
		DDD	DDE	DDD	DDE
Collections	4.96	1.06	1.67	4.68	2.97
JBoss-Cache	65.58	31.25	31.76	2.10	2.06
Lucene	11.02	3.57	6.12	3.09	1.80
Mina	0.26	0.17	0.20	1.53	1.30
Pool	1.43	1.04	1.04	1.38	1.38
Sysunit	17.67	0.35	0.45	50.49	39.27
JSR-166 TCK	15.20	9.56	9.56	1.59	1.59
Geometric Mean				3.39	2.76

Table 3.4: Test Execution Time (DDD/DDE: deadlock detection disabled/enabled)

numbers of orderings removed by cycles removal (CR), the number of orderings removed by transitive reduction (TR), and the number of orderings removed due to low confidence (LC). As it can be seen, the minimization phase does not remove many orderings. However, it is important to remove the orderings it does remove. For example, without removing the cycle for JBoss-Cache, not only would inference have a lower precision but it would also produce a schedule that is unrealizable.

3.5.3 Performance

Table 3.4 shows the execution times of the 198 original, sleep-based tests and the corresponding IMUnit tests (for IMUnit, with deadlock detection both disabled and enabled). We ran the experiments on an Intel i7 2.67GHz laptop with 4GB memory, using Sun JVM

1.6.0_06 and AspectJ 1.6.9. Our goal for IMUnit is to improve readability, modularity, and reliability of multithreaded unit tests, and we did not expect IMUnit execution to be faster than sleep-based execution. In fact, one could even expect IMUnit to be slower because of the additional code introduced by the instrumentation and the cost of controlling schedules. It came as a surprise that IMUnit is faster than sleep-based tests, on average 3.39x. Even with deadlock detection enabled, IMUnit was on average 2.76x faster. This result is with the sleep durations that the original tests had in the code.

Note that the time measurements shown are for one execution of each test, which follows one possible interleaving that satisfies the specified schedule. If the tests were explored (see Section 2.2) for more interleavings, the testing time would be much higher. Moreover, note that sleep-based tests cannot even be easily explored using tools that control schedules (see Sections 2.2.1 and 2.2.2) because of the interplay of these tools and real time. In contrast, IMUnit tests can be explored using such tools (with small adjustments).

We also compared the running time of IMUnit with MultithreadedTC on a common subset of JSR-166 TCK tests that the MultithreadedTC authors translated from sleep-based to tick-based [87]. For these 129 tests, MultithreadedTC was 1.36x faster than IMUnit, MultithreadedTC takes 7.07seconds to run, while IMUnit takes 9.66seconds. Although MultithreadedTC is somewhat faster, it has a much higher migration cost, and in our view, produces test code that is harder to understand and modify than the IMUnit test code. Moreover, we were surprised to notice that running MultithreadedTC on these tests, translated by the MultithreadedTC authors, can result in some failures (albeit with a low probability), which means that these MultithreadedTC tests can be unreliable and lead to false positives in test runs.

Chapter 4

CAPP Technique

This chapter presents the contributions of the Change-Aware Preemption Prioritization (CAPP) technique, which was developed with the aim of improving the exploration of multithreaded regression tests. This chapter is organized as follows. Section 4.1 presents our running example. Section 4.2 introduces the CAPP technique and presents its details. Section 4.3 presents our evaluation of CAPP and discusses the implications of the evaluation results.

4.1 Example

We first illustrate how CAPP works through an example of a real code evolution of Apache Mina [11]. Figures 4.1(a) and 4.1(b) show code snippets from two consecutive revisions of the `ProtocolCodecFilter` class in Mina. In the newer revision, 912149, developers inlined the invocation of the method `flushWithoutFuture` into the method `filterWrite` and further changed the loop condition to use the predicate `!bufferQueue.isEmpty()`. While performing these changes, the developers also removed the `null` check for `encodedMessage` (line 11 in Figure 4.1(a)). These changes, in fact, introduce a fault caused by an atomicity violation: if a preemption (preemptive context switch) occurs after a thread has checked `!bufferQueue.isEmpty()` and before it calls `bufferQueue.poll()`, another thread could remove elements from the `bufferQueue`, and `encodedMessage` could be assigned `null`, which results in a `NullPointerException`. This issue was reported in Mina's issue tracking system (DIRMINA-803 [11]) and corrected since then.

```
1  public void filterWrite(NextFilter nextFilter,
2                          IoSession session,
3                          WriteRequest writeRequest)
4                          throws Exception {
5   ...; flushWithoutFuture(); ...
6  }
7  public void flushWithoutFuture() {
8   Queue bufferQueue = getMessageQueue();
9   for (;;) {
10   Object encodedMessage = bufferQueue.poll();
11   if (encodedMessage == null) { break; }
12   // Flush only when the buffer has remaining.
13   if (!(encodedMessage instanceof IoBuffer) ||
14   ((IoBuffer) encodedMessage).hasRemaining()) {
15    ...
16    nextFilter.filterWrite(session, writeRequest);
17   }
18  }
19 }
```

(a) Code snippet from Mina revision 912148

```
1  public void filterWrite(NextFilter nextFilter,
2                          IoSession session,
3                          WriteRequest writeRequest)
4                          throws Exception {
5   ...
6   Queue bufferQueue = getMessageQueue();
7   while (!bufferQueue.isEmpty()) {
8    Object encodedMessage = bufferQueue.poll();
9    // Flush only when the buffer has remaining.
10   if (!(encodedMessage instanceof IoBuffer) ||
11   ((IoBuffer) encodedMessage).hasRemaining()) {
12    ...
13    nextFilter.filterWrite(session, writeRequest);
14   }
15  }
16  ...
17 }
```

(b) Code snippet from Mina revision 912149

```
1  class FilterWriteThread extends Thread {
2   int result = 0;
3   ...
4   public void run() {
5    try {
6     pc.filterWrite(nextFilter, session, writeRequest);
7     ...
8     } catch (Exception e) {
9      e.printStackTrace(); result = 1;
10    }
11   }
12 }
13 @Test
14 public void testFilterWriteThreadSafety() {
15  ...
16  FilterWriteThread fwThread1 = new FilterWriteThread(...);
17  FilterWriteThread fwThread2 = new FilterWriteThread(...);
18  fwThread1.start(); fwThread2.start();
19  fwThread1.join(); fwThread2.join();
20  assertEquals(0, fwThread1.result);
21  assertEquals(0, fwThread2.result);
22 }
```

(c) Multithreaded regression test for Mina

Figure 4.1: Example Evolution and Multithreaded Regression Test for Mina

Figure 4.1(c) shows a multithreaded test that exercises the changed code. This test creates two threads that call the faulty `filterWrite` method. This test has many possible schedules, and every time the code evolves, ideally the test should be explored for all the (non-equivalent) possible schedules to ensure there is no regression. Note that this test

does not specify any IMUnit schedules, which is equivalent to specifying a schedule with no orderings. Even if the test did specify a non-empty IMUnit schedule, it would still have to be explored for the (smaller) set of thread schedules/interleavings represented by the IMUnit schedule. Regardless of which schedules are specified, when a Mina developer re-explores this test after making the change to revision 912149, the sooner the test reveals the `NullPointerException`, the easier it is for the developer to debug [93].

Many promising techniques have recently been developed to improve the exploration of tests for a single version of code. One such promising technique is known as *iterative context bounding*, which is implemented in the CHESS tool [78]. CHESS prioritizes exploration according to the number of preemptive context switches between threads. The main idea is to explore the schedules that consist of a smaller number of context switches first, because many concurrency errors often manifest in such schedules. However, despite using such advanced exploration techniques to reduce the number of schedules, many different schedules need to be executed. Specifically, in this example, using the basic *change-unaware*, iterative context bounding exploration (with a bound of 2) *requires exploring 58 schedules* before the fault is revealed. In contrast, using *CAPP requires exploring as few as 5 schedules* before the fault is revealed, which is substantially faster.

CAPP can reduce the exploration required to reveal a fault by inferring the impact of code changes and prioritizing the exploration of schedules that perform preemptions at *Impacted Code Points (ICPs)*. CAPP uses different kinds of ICPs based on changed code lines/statements, methods, and classes, and the impact of these changes onto fields. Section 4.2.1 describes in detail how CAPP infers different kinds of ICPs. For example, in Mina revision 912149, CAPP marks as changed the highlighted lines in Figure 4.1(b). By analyzing the abstract syntax tree (AST) corresponding to these changed lines, CAPP infers that the method `filterWrite` and the class it belongs to are impacted by the changes. Moreover, while no fields are being directly accessed within the changed lines, CAPP analyzes the methods being invoked directly from the changed lines (in this case the methods `isEmpty`

and `poll`) and infers that the fields `head` and `tail` in the `Queue` class are also impacted by the change. Using ICPs like these, CAPP prioritizes the exploration of the multithreaded test to focus on changes and thus reveal the fault with substantially lesser exploration.

CAPP can use various heuristics to identify and prioritize change-impacted preemptions based on the set of collected ICPs. Section 4.2 describes in detail the 14 different heuristics that we propose. These heuristics are expected but not guaranteed to reveal the faults faster than the change-unaware exploration. Section 4.3 presents our empirical evaluation of the heuristics. For example, the heuristic Line On-stack All, which prioritizes the exploration of schedules that encounter states where *all* enabled threads are executing changed-impacted *lines* (such as lines 6-15 in Figure 4.1(b)), revealed the fault in just 5 schedules in this test. As another example, the heuristic Field Some, which prioritizes the exploration of schedules that encounter states where *some* enabled threads are accessing change-impacted *fields* (such as `head` and `tail`), revealed the fault in 19 schedules.

4.2 Technique

Change-Aware Preemption Prioritization consists of two main parts: static collection of a set of Impacted Code Points (ICPs) and dynamic prioritization of the exploration of schedules that perform preemptions at the collected ICPs. We first present collection of ICPs and then present the algorithm for prioritization of schedules.

4.2.1 Collecting Impacted Code Points

Collecting ICPs is similar to change-impact analysis [82,89,98]. 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 unidentified points

will still be explored, but 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 changes on the communication among threads, i.e., 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 `java.util.concurrent.ConcurrentLinkedQueue` class 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 [100]) is used to collect a set of lines that have been changed between the two versions. 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.

1. Any partial ICPs with changed lines that affect a synchronized region (e.g., adding or removing the `synchronized` keyword to or from methods, changing the scope of a `synchronized` block, etc.) are expanded to include the entire region (method/block). For example, if line 325 in `filterWrite()` were to belong to some `synchronized` block from lines 320 to 330, additional ICPs are added for all those lines.

2. 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.

3. 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 does not explicitly include any (changed) lines. Indeed, it encodes that *any* access to the field is potentially relevant and not only the accesses within the changed code.

4. Additional change-impacted field ICPs are collected by determining the read- and write-sets [92] 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.

4.2.2 Algorithm

CAPP uses the statically collected ICPs to dynamically prioritize the exploration of a multi-threaded regression test. Figure 4.2 shows the pseudo-code of the algorithm used to perform the prioritized exploration (note that this algorithm is an extension of the basic exploration algorithm that was presented in Section 2.2). The algorithm takes as inputs the test to be explored and a set of ICPs. The algorithm also has two parameters—prioritization mode and ICP match mode—that identify which heuristic to use (or none if `BASIC`). The possible values for these modes are listed at the top and are explained later in this section.

The algorithm uses the `Transition` pair to represent a transition that consists of a `State` and a `Thread` that can be executed in that state. The main data structures of the algorithm are `toExplore` and `nextToExplore`, which are both sets of transitions, to be explored either in the current iteration or in the next iteration, respectively. Typically these structures

```
1  // Parameters
2  enum PrioritizationMode { BASIC, ALL, SOME }
3  enum ICPMatchMode { CLASS, CLASS_ON_STACK, METHOD, METHOD_ON_STACK, LINE,
4                      LINE_ON_STACK, FIELD }
5  PrioritizationMode pMode; ICPMatchMode mMode;
6  // Exploration state
7  class Transition { State state; Thread thread; }
8  Set⟨Transition⟩ toExplore; Set⟨Transition⟩ nextToExplore; Set⟨State⟩ explored;
9  // Inputs
10 Test test; Set⟨ICP⟩ impactedCodePoints;
11 PassOrFail explore() { initializeExploration(); return performExploration(); }
12 void initializeExploration() {
13   State s_init = initial state for test; toExplore = {Transition(s_init, t) | t ∈ enabledThreads(s_init)};
14   nextToExplore = explored = {};
15 }
16 PassOrFail performExploration() {
17   while (toExplore ≠ {}) {
18     Transition current = pickOne(toExplore); toExplore = toExplore − { current }; restore current.state;
19     State s' = execute current.thread on current.state;
20     if (s' ∉ explored) {
21       if (s' is errorState) return FAIL;
22       Set⟨Transition⟩ enabled = {Transition(s', t') | t' ∈ enabledThreads(s')};
23       enabled = enabled − trans that satisfy some pruning condition such as partial order reduction;
24       Set⟨Transition⟩ prioritized = partitionPrioritized(enabled, current.thread);
25       toExplore = toExplore ∪ prioritized; nextToExplore = nextToExplore ∪ ( enabled − prioritized );
26       if (STATEFUL) { explored = explored ∪ { s' }; }
27     }
28     if (toExplore == {}) { toExplore = nextToExplore; nextToExplore = {}; }
29   }
30   return PASS;
31 }
32 Set⟨Transition⟩ partitionPrioritized(Set⟨Transition⟩ trans, Thread current) {
33   if (pMode == BASIC) { return trans; } // prioritization not performed
34   if (∄ t ∈ trans : t.thread == current) { return trans; } // preemption not possible
35   Set⟨Transition⟩ impacted = matchICPs(trans);
36   if (pMode == ALL) {
37     if (impacted == trans) { return trans; } else { return { pickOne(trans − impacted) }; } }
38   else // (pMode == SOME) {
39     if (impacted ≠ {}) { return trans; } else { return { t ∈ trans | t.thread == current }; } }
40 }
41 Set⟨Transition⟩ matchICPs(Set⟨Transition⟩ trans) {
42   Set⟨Transition⟩ matches = {};
43   for (tran ∈ trans) {
44     Instruction pc = tran.thread.pc; StackTrace st = tran.thread.stackTrace;
45     for (icp ∈ impactedCodePoints) {
46       if ((mMode == CLASS ∧ pc.cls == icp.cls) ∨ (mMode == CLASS_ON_STACK ∧ icp.cls ∈ st)
47       ∨ (mMode == METHOD ∧ pc.⟨cls, meth⟩ == icp.⟨cls, meth⟩)
48       ∨ (mMode == METHOD_ON_STACK ∧ pc.⟨cls, meth⟩ ∈ st)
49       ∨ (mMode == LINE ∧ pc.⟨cls, meth, ln⟩ == icp.⟨cls, meth, ln⟩)
50       ∨ (mMode == LINE_ON_STACK ∧ pc.⟨cls, meth, ln⟩ ∈ st)
51       ∨ (mMode == FIELD ∧ pc.instr is fieldInstr ∧ pc.⟨cls, fld⟩ == icp.⟨cls, fld⟩))
52         matches = matches ∪ { tran };
53     }
54   }
55   return matches;
56 }
```

Figure 4.2: Exploration Prioritization Algorithm

would be stacks (for the depth-first search strategy), queues (for the breadth-first search strategy), or priority queues (for search strategies like iterative context bounding that use other prioritization in addition to CAPP). Our algorithm is orthogonal to the search strategy and does not presume any particular search strategy. The algorithm also works for both stateful exploration (where `explored` tracks the explored states) and stateless exploration. For example, our ReEx tool applies the iterative context bounding prioritization strategy from CHESS [78].

The algorithm starts by initializing the data structures. The `toExplore` set is initialized with the enabled transitions of the initial state of the test, and the `nextToExplore` set is initialized to the empty set. The main exploration loop starts after the initialization and continues as long as `toExplore` is not empty. In each iteration of the main loop, a transition is selected and removed from the `toExplore` set. The state of the selected transition is reestablished (e.g., by restoring the state in JPF or re-executing the code in ReEx), and the thread of the transition is executed on the state to obtain the next state, `s'`. The algorithm then obtains the transitions that are enabled in `s'`. At this point, a selection criteria can be used to remove some enabled transitions from the enabled set.

The core part of the algorithm is the call to the function `partitionPrioritized` in line 24. This function partitions the enabled transitions into those that CAPP prioritizes for the current iteration and those it postpones for the next iteration, which it adds to the `toExplore` and `nextToExplore` sets, respectively. The partitioning is configured by the prioritization mode and the ICP match mode, described in Section 4.2.2. At the end of the main exploration loop, the algorithm checks whether the `toExplore` set is empty; if so, the transitions from the `nextToExplore` set are moved into the `toExplore` set to begin the next iteration of the CAPP exploration. This is repeated until the entire state space has been explored. However, it would be also possible to stop the loop after one or more iterations, which would result in selection rather than prioritization of schedules.

Modes

The algorithm takes two parameters, for the prioritization mode and for 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 ignore the threads that are not executing change-impacted code until they reach such code (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, as shown in line 34. 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 several 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 CAPP can be instantiated. Section 4.3 presents our evaluation of all 14 heuristics. 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.

4.3 Evaluation

The motivation behind CAPP is to reduce the exploration required to detect multithreaded regression faults. We designed and performed experiments to determine whether CAPP heuristics can indeed reduce the exploration cost to detect such faults. We also compare the heuristics and analyze their effectiveness across stateful/stateless exploration and default-/randomized search orders. The evaluation was conducted on 15 multithreaded faults from a diverse set of Java programs. We next present the implementations of CAPP that we use

in the experiments, the artifacts on which we performed the experiments, the experimental setup, measures used for the comparison, analysis of the results, and threats to validity.

4.3.1 Implementations

We implemented CAPP, along with all its heuristics, in two frameworks for systematic exploration of multithreaded Java programs, Java PathFinder (JPF) and ReEx. As described in Section 2.2.1, JPF is a widely used, stateful model checker for Java programs [103]. We implemented CAPP in JPF by customizing the existing `SimplePrioritySearch` to prioritize the search using CAPP. As described in Section 2.2.2, ReEx is a stateless exploration framework for Java programs that we developed using bytecode instrumentation. We developed a custom `SchedulingStrategy` in ReEx to control the search order using CAPP. The basic search strategy in JPF is (unbounded) depth first, while in ReEx it is the iterative context bounded from CHESS (with bound of 2) [78].

4.3.2 Artifacts

We conducted our experiments on a set of 15 multithreaded faults in Java programs. Table 4.1 provides more information about each of the faults and the programs in which those faults were detected. For each of the faults, we collected two versions of the program, a *correct* version without the fault and a *buggy* version with the fault. For each of the faults, we also collected a multithreaded regression test that passed on the correct version and failed on the buggy version. For many of the programs such a test was included in the test suite. In cases where such a test was not available, we created the appropriate test based on the information gained from the corresponding bug report. Further, some of the tests provided with the programs could be configured with the number of threads to be used. In such instances, we used a small number of threads that revealed the fault. This is in line with standard practice; developers are usually encouraged to write simple unit tests that check a

particular property. It is also the rationale behind techniques like CHESS [78] which detects faults with the smallest number of possible preemptions.

The statistics shown in Table 4.1 are for the buggy version of the programs on which the experiments were conducted. The correct versions were only used to obtain the initial diffs from which the ICPs for the buggy version were computed. The first 8 programs and their faults were obtained from the Software Infrastructure Repository (SIR) [35]. Each of these programs had one fault. We asked two graduate students to fix these faults to obtain the correct versions. The remaining 7 faults were obtained from various open-source Java projects. The correct and buggy versions for these faults were obtained from the respective project's source repository. The programs vary in size from 52 to 54,872 lines of code, and the size of the changes between the buggy and fixed versions ranges from 2 to 201 ICPs. The table also shows the type of error that was detected in each program. The assertion violations detected were all caused by data races or atomicity violations.

4.3.3 Setup

We conducted two sets of experiments with each implementation of CAPP. The first set of experiments measure the savings in terms of exploration cost that is achieved by using the CAPP heuristics compared to the basic exploration strategy of the respective exploration framework. Note that the basic exploration strategy naturally imposes a particular default search order on one exploration, e.g., in JPF depth-first strategy explores transitions enabled from a state in the order of the thread id of the transitions. However, a previous study has shown that exploration savings attributed to heuristics are often a function of the search order rather than the heuristic itself [37]. Therefore, to evaluate the effect of the search order, we conducted a second set of experiments with randomized search orders. For these experiments, we chose 50 random seeds and, for each seed, performed a randomized depth-first search similar to PRSS [36] for basic and heuristic explorations to compare the distribution of the exploration costs across all the seeds. In total we performed 19,890 explorations which

	Source	Error	#Threads	#Classes	#Methods	SLOC	#ICP
Airline	[102]	Assertion violation	6	2	18	136	6
Allocation	[102]	Assertion violation	3	3	22	209	5
BoundedBuffer	[102]	Deadlock	9	5	10	110	2
BubbleSort	[102]	Assertion violation	4	3	15	89	4
Deadlock	[102]	Deadlock	3	4	4	52	3
ReadersWriters	[102]	Deadlock	5	6	19	154	2
ReplWorkers	[102]	Deadlock	3	14	45	432	2
RAXextended	[102]	Assertion violation	6	11	23	166	2
Groovy	[97]	Deadlock	3	607	6399	54872	60
Lang	[12]	Assertion violation	3	215	4422	48369	3
Mina	[11]	Assertion violation	3	341	3188	34804	36
Pool1	[13]	Assertion violation	3	51	688	10042	148
Pool2	[14]	Assertion violation	3	35	371	4473	201
Pool3	[15]	Deadlock	2	51	706	10802	29
Pool4	unreported fault	Deadlock	3	51	705	10783	47

Table 4.1: Subject Regression Faults and Programs Statistics

required about a month of computing time to complete.

Note that not all of the artifacts could be used for both frameworks. Specifically, Mina could not be explored using JPF since it uses networking libraries that are currently not modeled by JPF. Also, ReplWorkers, RAXextended, and ReadersWriters could not be explored using ReEx since they are reactive programs where a re-execution/schedule can potentially run infinitely. ReEx currently does not support exploring such programs.

4.3.4 Measures

Because the experiments were conducted on multiple computers with different configurations (some experiments were performed on compute clusters), we do not measure exploration cost in real time. This is consistent with previous studies on exploration costs [36, 37, 106]. Instead, we measure the costs of exploration in terms of the *number of transitions (new states + visited states - 1) for JPF* and the *number of schedules (re-executions) for ReEx*. Recently, we conducted and published a systematic study that establishes that these metrics are strongly correlated with real time [58]. Note however that one of the findings of that study was that real time measurements from clusters could be useful. So in future studies we intend to measure and present real time results from clusters as well.

Note that CAPP prioritization does increase somewhat the per-transition or per-schedule real time cost compared to no prioritization, because CAPP checks for an ICP match (function `matchICPs` in Figure 4.2). However, for all modes without `ON_STACK`, this check can be rather cheap as it only compares the current program counter with a set of collected ICPs. In fact, the info can be statically pre-computed and each bytecode labeled with a boolean that indicates whether it is an ICP or not; the dynamic match then just checks the value of one boolean variable. For the modes with `ON_STACK`, the check is more expensive as it needs to maintain a stack of values and to compare the current program counter information with those values. Our current CAPP prototypes do not optimize these checks but follow

the pseudo-code in Figure 4.2 fairly directly. Another additional cost of CAPP over no prioritization is the static analysis for collecting ICPs. Yet again, our prototype does not attempt to minimize this cost, but it can be made rather negligible compared to the cost of exploration of numerous schedules for many multithreaded tests.

4.3.5 Results for Default Search Order

Tables 4.2 and 4.3 present the results of the first set of experiments measuring the savings in exploration cost that can be achieved by using various CAPP heuristics with the default search order. The second columns of the tables show the number of transitions/schedules required to detect the fault using the basic, change-unaware exploration provided by JPF/ReEx. The following columns show the speedup (if greater than 1.0) or slowdown (if less than 1.0) obtained by the various CAPP heuristics compared to the basic exploration. The highest speedup achieved for the detection of each fault is followed by a symbol '†'. The last rows show the geometric mean of the speedups achieved by the heuristics. Recall from Section 4.2 that the heuristic acronyms are built using the ICP match mode and the prioritization mode that define the heuristic. For example the Line On-stack Some (LOS) heuristic uses the `LINE_ON_STACK` ICP match mode and the `SOME` prioritization mode. The heuristics are referred to by their acronyms in all the tables. We use the results presented in these tables to address the first three research questions presented in Section 1.3.

RQ1: Exploration Cost

For stateful JPF exploration, the average reductions in exploration cost range from 1.0x for COS and MOS to 2.7x for MA, with the only average cost increases being 0.8x for LOS. For stateless ReEx exploration, the average reductions range from 1.5x for COS to 5.3x for LOA, with no average cost increases. Looking at individual faults, all the heuristics reduce cost in the detection of majority of the faults. For stateful JPF exploration, the greatest speedup of 37.8x was obtained by COA for detecting the RAXextended fault, and the greatest slowdown

	Basic	CA	CS	COA	COS	MA	MS	MOA	MOS	LA	LS	LOA	LOS	FA	FS
Airline	1328	1.0	1.0	1.0	1.0	1.0	0.9	1.0	1.0	1.0	0.9	1.0	1.0	1.3†	1.0
Allocation	1573	1.1	1.0	1.1	1.0	1.1	1.0	1.1	1.0	1.0	0.4	1.0	0.4	4.1†	1.0
BoundedBuffer	1391	1.4	1.1	1.4	1.0	1.4	1.1	1.4	1.0	1.8†	1.5	0.9	1.5	1.8†	1.5
BubbleSort	3880	1.2	1.0	1.2	1.0	0.9	1.1	2.6	1.0	0.9	1.1	2.6	1.0	3.2†	1.1
Deadlock	81	1.0	1.0	0.6	1.0	1.0	1.0	1.0	1.0	1.0	1.5	1.0	1.5	3.5†	1.8
ReadersWriters	1545	1.1	0.7	1.0	1.0	3.4†	0.3	3.0	0.3	1.3	0.4	1.3	0.4	3.3	0.2
ReplWorkers	5003	0.8	1.2	7.4	1.2	7.0	1.3	6.8	1.2	7.9†	4.8	7.9†	4.8	2.8	4.8
RAXextended	31987	0.6	2.9	37.8†	0.8	0.6	2.9	0.6	0.8	1.1	1.9	1.1	0.5	1.7	5.9
Groovy	6721	11.6†	1.0	11.6†	1.0	11.6†	1.0	11.6†	1.0	11.6†	1.0	11.6†	1.0	11.6†	1.0
Lang	1268	1.5	1.1	1.5	1.1	1.5	1.1	1.5	1.1	1.7	1.8†	1.7	1.8†	1.5	1.1
Pool1	2978	6.6	2.0	2.8	1.0	7.2†	2.3	2.8	1.0	1.2	2.2	1.2	1.9	1.0	0.7
Pool2	5077	6.4	1.4	3.1	1.0	8.7	4.9	3.1	1.0	29.4	6.7	35.5†	1.1	1.4	2.7
Pool3	507	2.6	1.2	1.6	1.0	1.8	1.2	1.8	1.0	2.6	0.2	2.6	0.02	5.1†	0.3
Pool4	9327	4.8	1.8	2.9	1.0	30.0†	8.3	15.0	2.8	0.8	1.5	1.0	1.4	0.8	9.8
Geom. Mean		1.9	1.2	2.4	1.0	2.7†	1.4	2.4	1.0	2.0	1.2	2.2	0.8	2.4	1.4

Table 4.2: Default Search Order Results for JPF († : max speedup for artifact)

	Basic	CA	CS	COA	COS	MA	MS	MOA	MOS	LA	LS	LOA	LOS	FA	FS
Airline	44312	1.0	1.0	1.0	1.0	13.4	1.0	13.5†	1.0	13.4	1.0	13.5†	1.0	2.2	1.0
Allocation	4101	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	0.4	0.6	0.4	0.6	512.6†	1.0
BoundedBuffer	1329	22.2	4.5	22.2	4.5	22.2	4.5	22.2	4.5	34.1†	4.5	34.1†	4.5	13.3	4.5
BubbleSort	5179	*	1.1	*	1.1	*	1.1	*	1.1	*	1.1	*	1.1	*	1.1
Deadlock	6	1.2	0.6	1.2	0.6	1.2	0.6	1.2	0.6	1.2	0.8	1.2	0.8	1.5	2.0†
Groovy	19	1.0†	0.4	1.0†	0.4	1.0†	0.4	1.0†	0.4	1.0†	0.4	1.0†	0.4	0.6	0.7
Lang	9	2.3	3.0†	2.3	3.0†	2.3	3.0†	2.3	3.0†	2.3	3.0†	2.3	3.0†	2.3	3.0†
Mina	58	0.6	3.1	0.9	2.9	0.7	2.6	0.6	3.9	0.7	2.6	11.6†	8.3	0.6	3.1
Pool1	6463	4.4	8.7	1.2	8.0	30.6	22.6	1.2	8.0	119.7†	113.4	10.6	22.0	1.3	8.0
Pool2	98	0.3	1.2	0.1	1.3	0.4	1.4	0.1	1.3	16.3†	3.6	12.3	1.6	2.3	8.2
Pool3	2	2.0†	1.0	1.0	1.0	2.0†	1.0	1.0	1.0	0.5	1.0	0.5	1.0	0.5	1.0
Pool4	13593	6.3	15.7	2.0	1.1	17.8	31.9	121.4	16.1	1.0	15.3	1.0	0.9	1.1	199.9†
Geom. Mean		3.1	1.9	2.3	1.5	5.1	2.1	3.8	1.9	5.1	2.4	5.3†	1.7	4.3	3.1

Table 4.3: Default Search Order Results for ReEx († : max speedup for artifact, * : 1294.8†)

of 0.02x was obtained by LOS for detecting the Pool3 fault. For stateless ReEx exploration, the greatest speedup of 1294.8x was obtained by the `ALL` prioritization mode based heuristics for detecting the BubbleSort fault, and the greatest slowdown of 0.1x was obtained by MOA for detecting the Pool2 fault.

The CAPP heuristics do reduce exploration costs on average, ranging from 1.0x to 5.3x. Only 1 instance (out of 28) increase costs on average with 0.8x.

RQ2: Comparison of Heuristics

For JPF, grouping the heuristics by the ICP match mode, `METHOD` heuristics perform the best, followed by `FIELD` heuristics, and only the `METHOD_ON_STACK` and `LINE_ON_STACK` heuristics have average slowdowns. For ReEx, `LINE` heuristics perform the best, followed by `FIELD` heuristics, with the worst being `CLASS_ON_STACK` heuristics. Grouping by prioritization mode, each `ALL` heuristic outperforms its corresponding `SOME` heuristic for both JPF and ReEx.

Heuristics based on the `FIELD` ICP match mode perform better than heuristics based on the other ICP match modes. Heuristics based on the `ALL` prioritization mode perform better than heuristics based on the `SOME` prioritization mode.

RQ3: Stateful vs Stateless Exploration

The CAPP heuristics achieve speedups for both stateful and stateless explorations, but on average the speedups are greater for stateless exploration. There are also a few other differences between the performance of the heuristics across stateful and stateless exploration. While MA (with 2.7x speedup) is the best heuristic on average for stateful exploration, LOA (with 5.3x speedup) is the best heuristic on average for stateless exploration. While stateful exploration has two heuristics with average slowdowns (MOS and LOS), none of

the stateless heuristics have average slowdown. Grouping the heuristics by prioritization mode, the `ALL` heuristics outperform the corresponding `SOME` heuristics for both stateful and stateless exploration. However, grouping heuristics by ICP match mode, `METHOD` is the best for stateful exploration, while `LINE` is the best for stateless exploration. `FIELD` performs consistently well for both stateful and stateless exploration.

The CAPP heuristics achieve greater reduction in exploration costs for stateless exploration. `ALL` and `FIELD` heuristics perform well for both stateful and stateless exploration.

While the default strategy in JPF is depth-first search (DFS), we also evaluated CAPP with the breadth-first search (BFS) strategy. The absolute numbers of transitions required to detect the faults were orders of magnitude larger for BFS than for DFS. However, BFS with the CAPP heuristics still performed better than Basic BFS and, in fact, achieved around twice as high average cost reduction than achieved for DFS with CAPP over Basic DFS.

4.3.6 Results for Randomized Search Order

Tables 4.4, 4.5, 4.6, and 4.7 show the results of the second set of experiments with 50 seeds that randomize the default search order [36]. For each fault, we show box plots for the randomized basic and randomized CAPP heuristic explorations, comparing the distributions of transitions/schedules that are explored to detect the fault. Each box plot shows the median, upper and lower quartile values, the max and min values not outside the 1.5 times inter-quartile range from their quartile values, and the outliers outside that range. Note that each y-axis is normalized such that the 100 mark is the median of the randomized basic exploration, and all other values are divided by that median. We also include a reference line (`Default`) that shows the number of transitions/schedules that the basic exploration with the (non-randomized) default search order explored to detect the fault (i.e., the values from the second column of tables 4.2 and 4.3); when the line is missing, it is above the

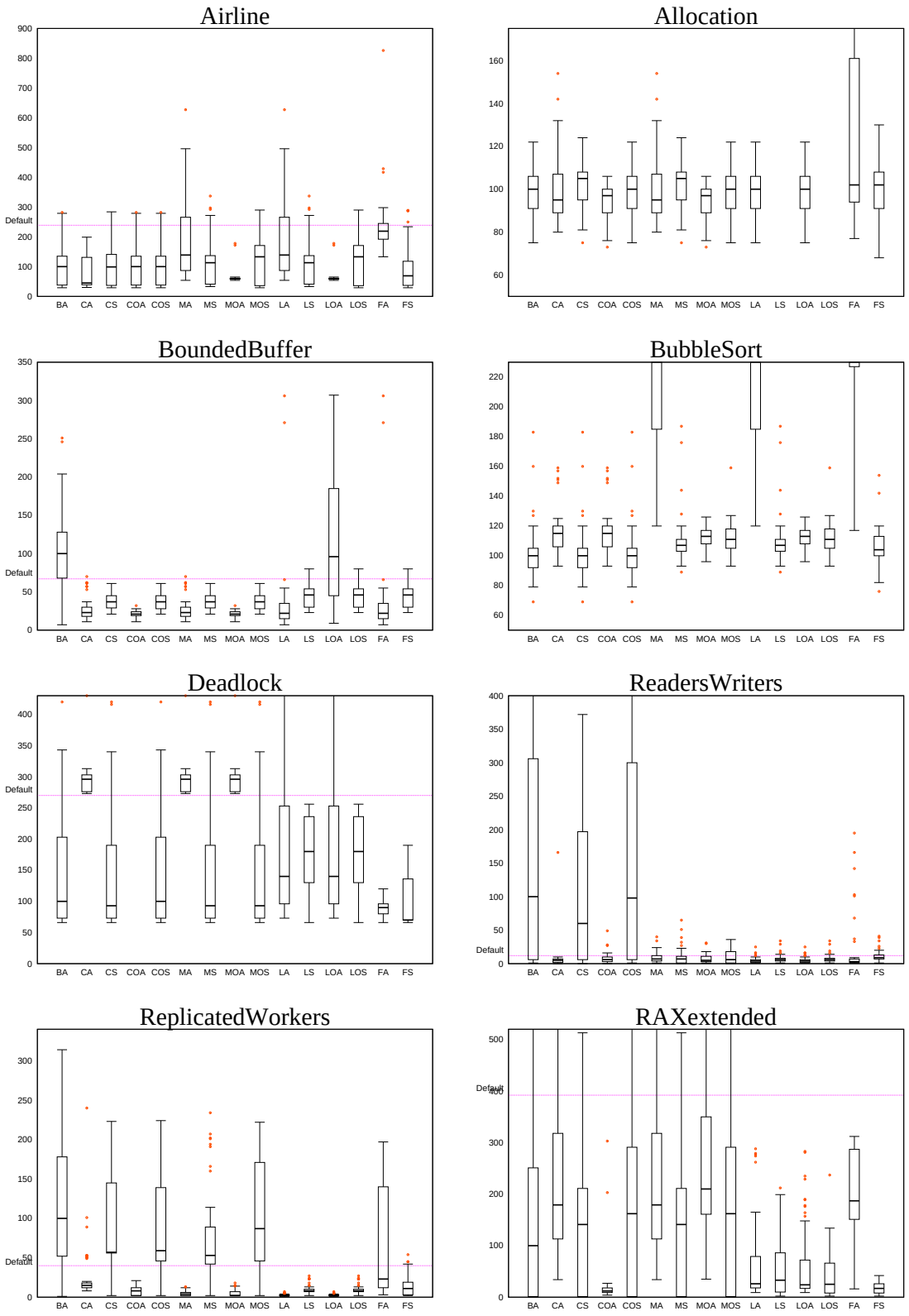

Table 4.4: Randomized Search Order Results for JPF - Part 1 of 2

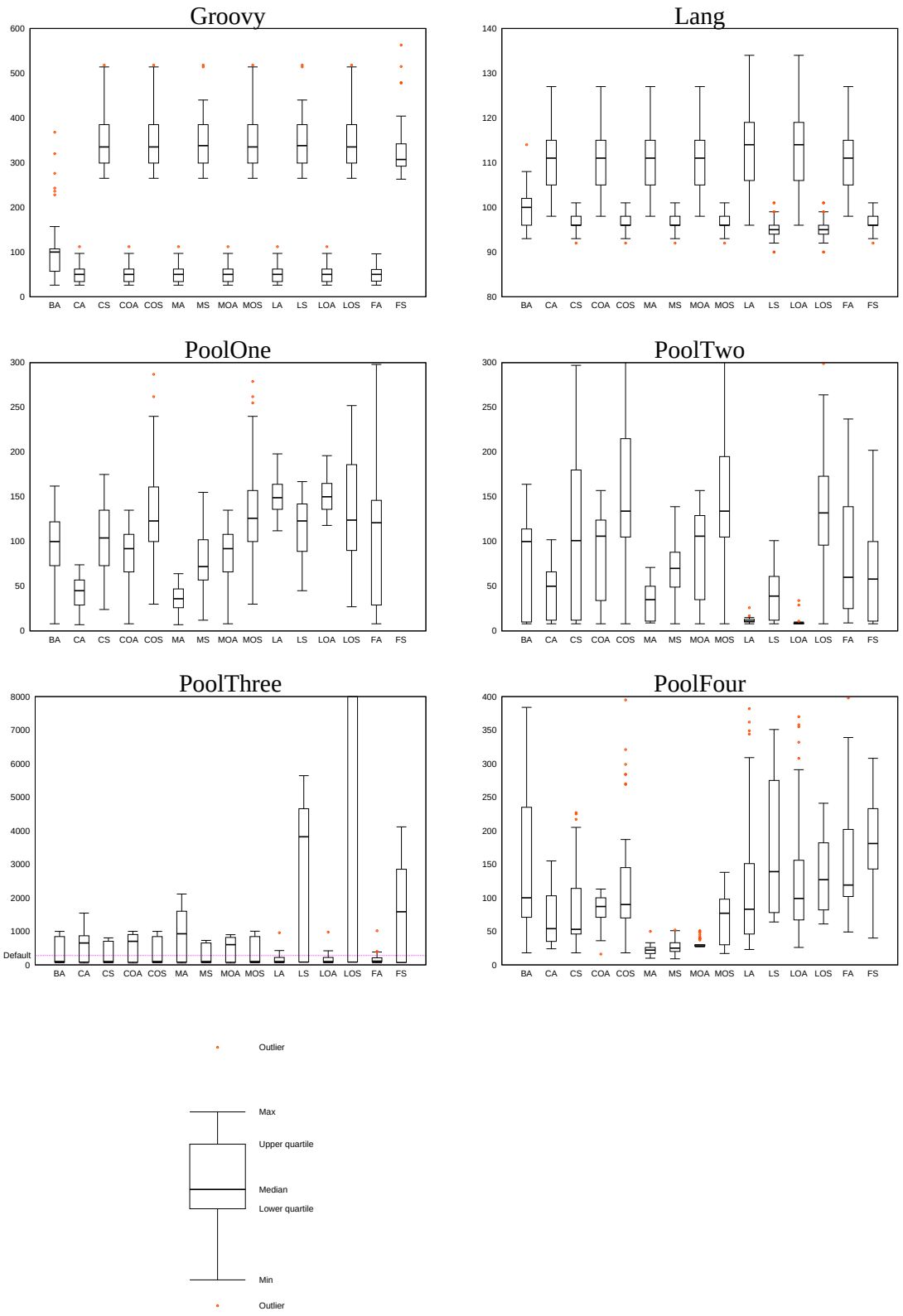

Table 4.5: Randomized Search Order Results for JPF - Part 2 of 2

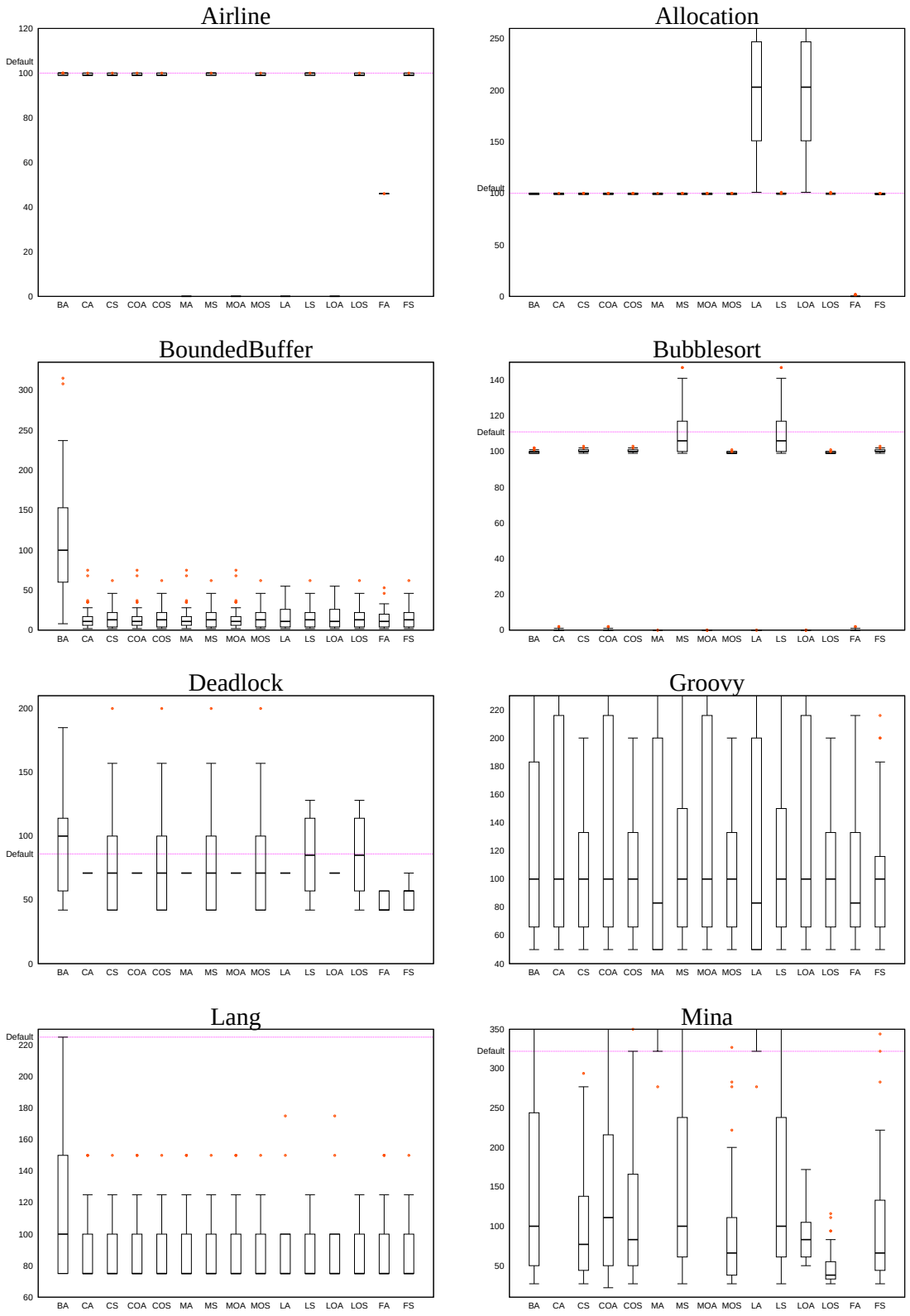

Table 4.6: Randomized Search Order Results for ReEx - Part 1 of 2

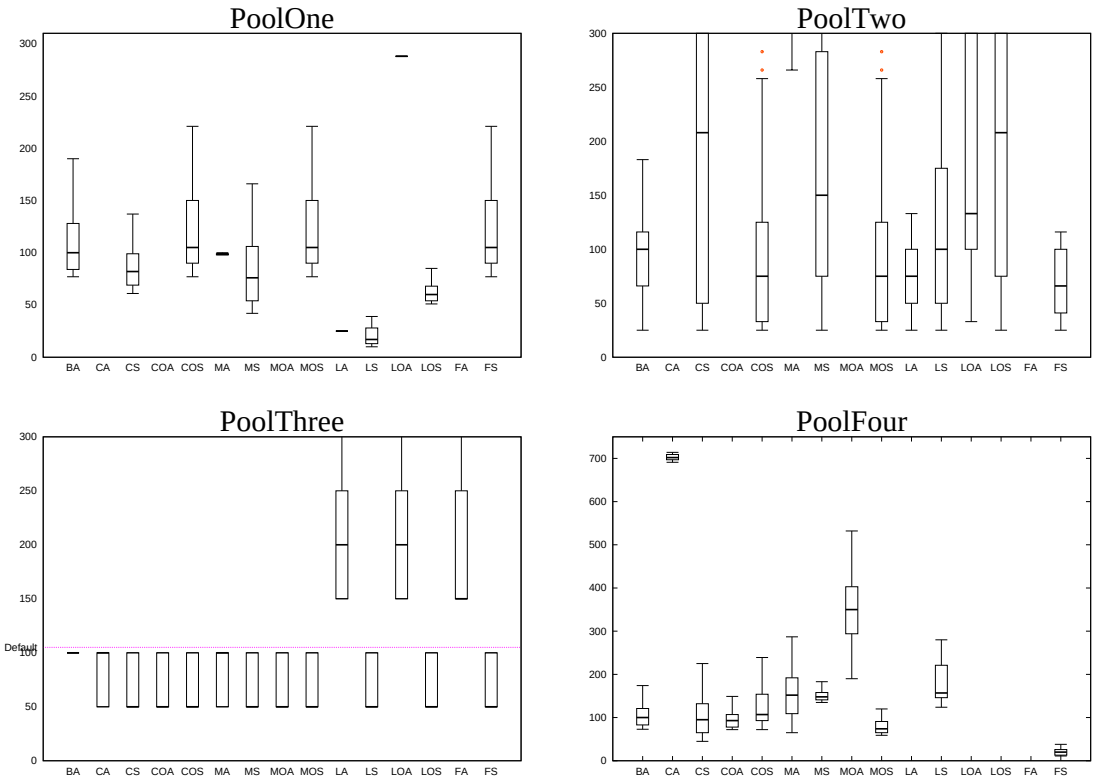

Table 4.7: Randomized Search Order Results for ReEx - Part 2 of 2

limit. Additionally, following the methodology recommended by Arcuri and Briand [16], we performed pairwise Mann-Whitney U tests and computed Vargha and Delaney's $\hat{A}_{12}$ non-parametric effect size measure to compare the random search order results distributions for each of the heuristics with basic. Table 4.8 shows the computed p-values and $\hat{A}_{12}$ values.

RQ4: Default vs Random Search Order

A heuristic can be considered to perform better than basic with high confidence if its p-value is less than 0.05 and its $\hat{A}_{12}$ is greater than 0.5. As noticed with the default search order, majority of the heuristics for both stateful JPF and stateless ReEx exploration do perform better than basic. The best heuristics for JPF (MA) and ReEx (LOA) with the default search order, continue to be the best for random search order. Grouping heuristics by the ICP match mode, `METHOD_ON_STACK` and `LINE_ON_STACK` heuristics perform well for JPF, and `LINE` heuristics perform well for ReEx, which is contrary to the default search order results.

Heuristic	JPF		ReEx	
	p-value	$\hat{A}_{12}$	p-value	$\hat{A}_{12}$
CA	0.0661	0.5306	0.0249	0.5346
CS	0.0655	0.5307	0.5653	0.4911
COA	0.0283	0.5365	0.0267	0.5342
COS	0.0607	0.5313	0.1095	0.4752
MA	0.0001	0.6411	0.0054	0.5430
MS	0.2181	0.5205	0.0947	0.5258
MOA	0.0001	0.6082	0.0025	0.5467
MOS	0.0161	0.5401	0.6015	0.4920
LA	0.0001	0.6299	0.0012	0.5499
LS	0.0703	0.5302	0.0012	0.4499
LOA	0.0001	0.6109	0.0001	0.5594
LOS	0.0351	0.5351	0.0001	0.4296
FA	0.0046	0.5471	0.4769	0.4890
FS	0.0007	0.5563	0.1770	0.4792

Table 4.8: Summary of Randomized Search Order Results

Grouping heuristics by prioritization mode, `ALL` heuristics continue to do better than `SOME` heuristics.

The results for random search order confirm all the results for default search order (e.g., in terms of best heuristics and best prioritization modes), except that the ICP match modes that are the best for default search order are not the best for random search order.

4.3.7 Discussion

The performance of the various CAPP heuristics can be explained based on their constituent prioritization mode and ICP match mode. Recall the Mina revision from Figure 4.1(b); the introduced fault is triggered by a preemption between a thread that is about to execute changed line 8 (which assumes that `bufferQueue` is not empty) and another thread that is about to execute an unchanged line within the `bufferQueue.poll()` call in line 8 (which could make `bufferQueue` empty). Since the failure inducing preemption is between a changed line and an unchanged line, `SOME` prioritization mode based heuristics that prioritize such pre-

emptions perform better for Mina compared to their `ALL` counterparts (Table 4.3). The only exception is LOA, which performs better than LOS. Note that the execution of unchanged lines within `bufferQueue.poll()` originates from changed line 8. Hence the changed line is on the stack trace when the unchanged lines are executed. So when the stack trace is included in the match, both lines that are part of the failure inducing preemption are considered to be change impacted. In such situations the `ALL` prioritization mode based heuristics perform better since they prioritize preemptions where both threads are executing change impacted locations. Another example of such a situation is the fault in ReadersWriters, which is triggered by a preemption between two threads that are both about to execute change impacted locations. So the `ALL` prioritization mode based heuristics that prioritize such preemptions perform better for ReadersWriters compared to their `SOME` counterparts (Table 4.2). The majority of faults that we evaluated were caused by preemptions between threads that are both executing change impacted locations. Hence, on average, the `ALL` prioritization mode based heuristics performed better than their `SOME` counterparts.

In terms of the ICP match mode, the `CLASS` match mode based heuristics generally perform worse than their lower granularity `METHOD`, `LINE` and `FIELD` counterparts. The `CLASS` match mode based heuristics end up prioritizing too many preemptions that do not induce faults and hence delay fault detection. Among the `METHOD`, `LINE` and `FIELD` match mode based heuristics, variance in performance is dependent on fault inducing preemption(s) i.e. whether it is sufficient to preempt between change impacted methods, lines or fields. Going back to the Mina fault, the changed location (Figure 4.1(b), line 8) that is part of the fault inducing preemption does not access any fields and the locations involved are in different methods, hence the `LINE` match mode based heuristics perform better (Table 4.3).

4.3.8 Threats to Validity

In this section we describe threats to the validity of our evaluation of CAPP.

Internal threats: We conducted our experiments using the default settings of JPF (version 5.0pre2) and ReEx systematic exploration frameworks. During the course of our study, we found two bugs in JPF, which we have corrected on our local copy. To the best of our knowledge, there are no other bugs in the frameworks that would affect our results. However, changing the settings could affect the results.

External threats: The artifacts that we perform our experiments on were collected from a variety of sources and are diverse in terms of the statistics that we show in Table 4.1. The artifacts that we collected from SIR [102] have all been used in previous experiments, and the other artifacts are from widely used open-source projects. However, we cannot guarantee that they form a representative sample of multithreaded Java programs. To mitigate the limitation of using one particular exploration framework and search order, we evaluated CAPP with two different exploration frameworks, with both default and randomized search orders, and with both DFS and BFS for JPF.

Construct threats: In our evaluation, we use the number of transitions (for JPF) and the number of schedules (for ReEx) as the measures for exploration cost instead of the real execution time. The reason for this was three fold. First, we performed our experiments across multiple computers with various hardware configurations, hence measuring real execution time across computers may not be a robust measure. Second, previous related studies [36, 37, 58, 106] also use abstract, system-independent measures, such as the number of new states, which have been shown to be strongly correlated with real time. Third, our CAPP prototypes do not optimize for speed as our goal was to evaluate the algorithms before focusing on the implementation.

Conclusion threats: The number of random explorations (50) that we performed for each artifact and heuristic may not be sufficient to accurately characterize real distribution of the random explorations.

Chapter 5

Related Work

This chapter presents an overview of the work related to the contributions of this dissertation.

Expressing schedules: ConAn [72, 73] and MultithreadedTC [88] introduce unit testing frameworks that allow developers to specify schedules to be used during the execution of multithreaded unit tests. However, the schedules in both frameworks are specified relative to a global clock (real time for ConAn and logical time for MultithreadedTC), which potentially makes it difficult to reason about the schedules. In contrast, IMUnit schedules are intuitively specified as orderings over events. Also, unlike IMUnit, neither framework supports automated migration of sleep-based tests. Chapter 3 presents a detailed description of all the advantages of IMUnit.

Enforcement of schedules: There has been some previous work on using formally specified sequencing constraints to verify multithreaded programs [99]. In this work, the specifications are over synchronization events with LTL-like constraints, and the verification ensures that the implementation is faithful to the specification. In contrast, IMUnit schedule specifications are used to enforce ordering between user-specified events while the system is tested by checking user-specified assertions for the enforced ordering. Carver and Tai [24] use deterministic replay for concurrent programs. LEAP [56] is a more recent system using a similar record-and-replay approach based on Java bytecode instrumentation to reproduce bugs. In comparison, our enforcement and checking mechanism targets user-specified schedules rather than replaying a previously observed execution.

Automated inference/mining of specifications: Work on automated mining of specifications for programs [4, 5, 23, 70] is related to our automated inference of events and schedules. However, most existing work focuses on mining API usage patterns/rules in a single-threaded scenario, while our techniques mine the intention of sleep-based tests, i.e., interesting events and event orderings across multiple threads.

Exception propagation and assertions: ConcJUnit [90] extends JUnit to propagate exceptions raised by child threads up to the main thread and also checks whether all child threads have finished at the end of a test method. ThreadControl [34] proposes a tool to ensure that assertions are performed without interference from other threads. These features could be added to the IMUnit framework; they are orthogonal to the main goal of IMUnit, which is to help developers express and enforce schedules in multithreaded unit tests in a reliable, efficient, modular, and intuitive manner.

Event-based behavioral abstractions: The language used in event-based behavioral abstractions [18, 67] is similar to the IMUnit schedule language. However, event-based behavioral abstractions are used primarily to perform post-mortem debugging of distributed systems, whereas IMUnit is used to test multithreaded programs. IMUnit allows developers to express schedules in tests and then enforces/checks the schedules dynamically during the execution of the test. The techniques using event-based behavioral abstractions do not perform any dynamic enforcement, they collect traces during the execution of a distributed system, and then perform post-mortem analysis/checks on the traces to help with debugging.

Two-level semantics and synchronizers: The notion of separating the specification of the local behavior of an object and the global behavior of an object (i.e., communication between objects) has been explored in the context of concurrent object-oriented programming [2] and actors [1]. The specification of schedules in IMUnit demonstrates many of the advantages of such an approach, including modularity. In IMUnit, sched-

ules are specified separately from the rest of the test, and the IMUnit schedule language shares some similarity with the language used to describe synchronizers [47, 48] for actors. Synchronizers were introduced to specify and enforce coordination properties over the distributed behavior of actors. In contrast, IMUnit schedules are used to test multithreaded programs.

Improving Regression Testing: Many techniques have been developed for improving regression testing of sequential code. Test selection [52, 105] techniques choose to run only a subset of tests on the new program version. The key challenge is to perform safe selection [91], i.e., guarantee that tests that are not selected will not reveal faults. Test prioritization [39, 40, 62, 71, 95, 98, 104, 110] reorders (all or only selected) tests to reveal faults faster, thus reducing the time that a developer has to wait to find failing tests. Impact analysis [82, 89, 98] finds (statically or dynamically) which code changes could affect which tests, thus aiding test selection or debugging by pointing out which changes could (not) lead to failing tests. These techniques work well for selecting/prioritizing *among* sequential tests, which are typically short running. However, multithreaded tests are typically long running because they are explored for many different schedules. Hence, when a regression test suite contains multithreaded tests, selecting/prioritizing schedules *within* one test becomes an issue. While the exploration of multiple tests can be easily parallelized, efficiently parallelizing a single exploration is very challenging (e.g., witness many years of the PDMC workshop series). CAPP prioritizes the exploration of schedules for a single multithreaded test to reduce the exploration required to detect a fault (if one exists).

Change-unaware testing techniques: There is also a rich body of work on testing multithreaded code but mostly with *change-unaware* techniques [20, 22, 29, 42, 43, 64, 78, 80, 84]. Most of these techniques conceptually prioritize (or select) schedules to be explored such that faults are found faster (or exploration finishes faster if there are no

faults), but the prioritization does not consider code changes. We believe that most of these techniques can be modified to be *change-aware* and that it would provide faster exploration when code evolves, but in this dissertation we focused on using CHESS [78] as an example stateless technique and JPF as an example stateful technique (with its underlying partial-order reductions). The original work on CHESS [78] showed that a large number of concurrent faults can be detected with a small number of preemptions, often up to two. The follow-up work on preemption sealing [17] employs a new scheduling strategy that only allows preemptions around certain program modules to enable modular testing, e.g., to speed up testing of applications that use reliable libraries. All these techniques focus on a *single* program version, while CAPP considers changes between two program versions to prioritize schedules, which results in faster detection of regressions.

Change-aware testing techniques: We know of only a few *change-aware* (also called incremental) techniques for systematic testing. Initial work in this area focused on control-intensive properties in model checking [28, 53, 74, 96], conceptually reusing results from one run to speed up the next run. The ISSE technique [69] also reuses results from one run to another but for data-intensive properties of sequential, non-deterministic code. The projects that are most recent and most related to this dissertation are regression model checking (RMC) [106] and improved mutation testing of multithreaded code [51]. Both projects reuse exploration of one program version to speed up the exploration of the next program version. However, these projects effectively focus on *selection* and attempt to only explore the states that behave differently after a change has been made. Our work differs in that (1) CAPP focuses on *prioritization*, (2) CAPP does not reuse the results from a previous exploration but only exploits changes and their impact, and (3) our evaluation includes real faults and not only mutants. It is likely that exploiting previous exploration can improve CAPP even

further, and that can be investigated in the future.

Other uses of evolution information: Our work provides faster rerunning of existing tests after code changes. But since changes often result in faults [79], a number of projects focus on other problems related to changes. For example, automated generation of new tests after code changes recently gained attention [61, 86]. BERT [61] also targets changes between two versions of a program. It generates test inputs and identifies behavior differences when executing different versions of programs. Also, leveraging information about changes can help in debugging [89, 109], and comparing successful and failed runs of multithreaded programs can help in debugging faults [26, 85].

Chapter 6

Conclusion and Future Work

Due to the non-determinism caused by thread scheduling, multithreaded programs are hard to test. Developers need the ability to express and enforce schedules in tests, and since every test can have many possible schedules, costly exploration is required. The contributions of this dissertation address these problems. IMUnit allows developers to express and enforce schedules in a reliable, intuitive, and modular fashion, and CAPP enables effective change-aware exploration of tests. We now present potential future work building upon our current contributions and results as described in Chapters 3 and 4:

Generation of IMUnit Tests: As mentioned previously, many tools have been developed to automatically detect bugs in multithreaded software by exploring/generating interesting schedules that may reveal bugs. Once a bug is found using such tools, ideally the developer should write an IMUnit test that captures the bug and add it to the regression test suite. It would be even better if the tool itself generates an IMUnit test that can reproduce the bug. The challenge will be to generate IMUnit schedules that are concise in terms of the number of orderings required to reliably reproduce the bug. Automated generation of such an IMUnit schedule would greatly improve the debugging experience for developers. IMUnit tests could also be generated automatically to exercise certain properties and build better test suites. Many such test generation techniques have been developed successfully for sequential programs [21, 32, 49, 76, 94, 101]. The recently introduced Ballerina technique [81], which generates multithreaded tests that expose multithreaded bugs, could possibly be improved by generating IMUnit schedules for the tests.

Preventing Multithreaded Regressions with IMUnit: When a multithreaded bug is fixed, the schedules expressed in the IMUnit test intended to reproduce the bug may no longer be feasible. Hence, the test may need to be discarded, reducing the effectiveness of the regression test suite. Further work is required to develop mechanisms to retain such IMUnit tests even after the bug has been fixed. One alternative could be to utilize the specified IMUnit schedules as a specification of schedules that should no longer be feasible, i.e., their enforcement should lead to a deadlock. In future runs of the test, the schedule enforcement will be expected to lead to a deadlock. If a deadlock does not occur, a test failure could be reported.

Automated Repair of IMUnit Schedules: As the system under test evolves, IMUnit tests may break and will need to be updated/repaired to capture the new requirements/behavior. Repairing an IMUnit test may entail updating the schedules that are being tested. This process could be automated using a refactoring based approach, where repairs would be automatically inferred and then presented to the developer for approval (similar to the automated repair of sequential tests as presented by ReAssert [33]).

Prescriptive Use of IMUnit Schedules: Multithreaded bugs like data races, deadlocks and atomicity violations usually manifest in schedules that developers have not thought about. On the other hand, IMUnit schedules written by developers represent schedules that developers have reasoned about and tested for expected behavior. Hence, the IMUnit schedules could potentially be utilized in a prescriptive manner, by enforcing them during production runs in order to restrict the execution to known/tested schedules. This would require overcoming many challenges in terms of determining which schedules are applicable during a particular production run and minimizing enforcement overhead. The Tern system [31] explores similar usage of schedules in multithreaded programs, by memoizing safe schedules and reusing them in later runs.

Furthermore, IMUnit could also be used by developers to specify and enforce the schedules that should be used during production runs, rather than just for testing. Such use of IMUnit schedules is similar to the concept of synchronizers [47, 48], which was introduced to coordinate the behavior of distributed actors [1].

Other Types of Multithreaded Tests: Using IMUnit, developers can write better multithreaded unit tests when they are aware of the schedule(s) to be tested. However, often developers may not be aware of specific schedules, but they still write multithreaded tests. One common instance of such tests are stress tests, where developers exercise a SUT with multiple threads and execute the test many times hoping that different executions follow different schedules to increase the chances of finding schedule relevant bugs. These tests usually assert very general properties (e.g., no crash or uncaught exception) and are unlikely to exercise a diverse set of schedules. To address this problem, systems like ConTest [38] have been developed to increase the effectiveness of stress tests. One could conduct a study on the prevalence of stress tests (and other types of multithreaded tests) in existing test suites and compare the utility of various types of multithreaded tests. The results of this study could provide guidance for future research.

Combining Selection and Prioritization: By prioritizing the exploration, CAPP is able to detect regression faults faster. However, if the evolution has not introduced any regression faults, CAPP will not achieve any reduction in exploration cost. In order to overcome this deficiency, one could explore the possibility of combining change-aware selection techniques similar to MuTMuT [50, 51] and Regression Model Checking [106] with CAPP.

Predicting Heuristics Performance: While all the CAPP heuristics achieve speedups on average, different heuristics performs best for different faults/evolutions. Section 4.3.7 presents some intuition as to why some heuristics perform better than others for par-

ticular faults. One could study the instances where particular heuristics perform significantly better than others in order to gain a deeper understanding of the underlying causes and possibly develop automated prediction techniques to decide which heuristic to use a priori.

Parallel Exploration using Multiple Heuristics: Researchers have previously explored techniques like PRSS [36] and Swarm [54], which take advantage of multiple diverse explorations by performing them in parallel. In PRSS the diverse explorations include randomized explorations with different seeds and in Swarm they include various exploration orders based on various heuristics. Since different CAPP heuristics work best for different programs, one could consider taking advantage of this diversity by performing explorations based on the different heuristics in parallel.

References

[1] Gul Agha, *ACTORS: A model of concurrent computation in distributed systems*, MIT Press, 1986.

[2] ———, *Concurrent object-oriented programming*, CACM (1990).

[3] A. V. Aho, M. R. Garey, and J. D. Ullman, *The transitive reduction of a directed graph*, SIAM Computing (1972).

[4] Rajeev Alur, Pavol Cerný, Madhusudan Parthasarathy, and Wonhong Nam, *Synthesis of interface specifications for Java classes*, POPL, 2005.

[5] Glenn Ammons, Rastislav Bodík, and James R. Larus, *Mining specifications*, POPL, 2002.

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

[7] ———, *Apache Commons Pool*, `http://commons.apache.org/pool/`.

[8] ———, *Apache Lucene*, `http://lucene.apache.org/`.

[9] ———, *Apache MINA*, `http://mina.apache.org/`.

[10] ———, *Apache River*, `http://river.apache.org/`.

[11] ———, *DIRMINA-803*, `https://issues.apache.org/jira/browse/DIRMINA-803`.

[12] ———, *LANG-481*, `https://issues.apache.org/jira/browse/LANG-481`.

[13] ———, *POOL-107*, `https://issues.apache.org/jira/browse/POOL-107`.

[14] ———, *POOL-120*, `https://issues.apache.org/jira/browse/POOL-120`.

[15] ———, *POOL-146*, `https://issues.apache.org/jira/browse/POOL-146`.

[16] Andrea Arcuri and Lionel Briand, *A practical guide for using statistical tests to assess randomized algorithms in software engineering*, ICSE, 2011.

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

[18] Peter Charles Bates, *Debugging heterogeneous distributed systems using event-based models of behavior*, ACM TOCS (1995).

[19] Kent Beck, *Test driven development: By example*, John Wiley & Sons, 2002.

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

[21] Chandrasekhar Boyapati, Sarfraz Khurshid, and Darko Marinov, *Korat: Automated testing based on Java predicates*, ISSTA, 2002.

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

[23] Jacob Burnim and Koushik Sen, *DETERMIN: Inferring likely deterministic specifications of multithreaded programs*, ICSE, 2010.

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

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

[26] Jong-Deok Choi and Andreas Zeller, *Isolating failure-inducing thread schedules*, ISSTA, 2002.

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

[28] Christopher L. Conway, Kedar S. Namjoshi, Denis Dams, and Stephen A. Edwards, *Incremental algorithms for inter-procedural analysis of safety properties*, CAV, 2005.

[29] Katherine Coons, Sebastian Burckhardt, and Madanlal Musuvathi, *Gambit: Effective unit testing for concurrency libraries*, PPoPP, 2010.

[30] Scott Cotton, *graphlib*, `http://www-verimag.imag.fr/~cotton/`.

[31] Heming Cui, Jingyue Wu, Chia-Che Tsai, and Junfeng Yang, *Stable deterministic multithreading through schedule memoization*, OSDI, 2010.

[32] Brett Daniel, Danny Dig, Kely Garcia, and Darko Marinov, *Automated testing of refactoring engines*, ESEC/FSE, 2007.

[33] Brett Daniel, Vilas Jagannath, Danny Dig, and Darko Marinov, *ReAssert: Suggesting repairs for broken unit tests*, ASE, 2009.

[34] Ayla Dantas, Francisco Vilar Brasileiro, and Walfredo Cirne, *Improving automated testing of multi-threaded software*, ICST, 2008.

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

[36] Matthew B. Dwyer, Sebastian G. Elbaum, Suzette Person, and Rahul Purandare, *Parallel randomized state-space search*, ICSE, 2007.

[37] Matthew B. Dwyer, Suzette Person, and Sebastian G. Elbaum, *Controlling factors in evaluating path-sensitive error detection techniques*, FSE, 2006.

[38] Orit Edelstein, Eitan Farchi, Evgeny Goldin, Yarden Nir, Gil Ratsaby, and Shmuel Ur, *Framework for testing multi-threaded Java programs*, Wiley CCPE (2003).

[39] Sebastian Elbaum, Alexey Malishevsky, and Gregg Rothermel, *Incorporating varying test costs and fault severities into test case prioritization*, ICSE, 2001.

[40] Sebastian Elbaum, Alexey G. Malishevsky, and Gregg Rothermel, *Test case prioritization: A family of empirical studies*, IEEE TSE (2002).

[41] K.F. Fischer, F. Raji, and A. Chruscicki, *A methodology for retesting modified software*, NTC, 1981.

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

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

[44] The Eclipse Foundation, *The eclipse development platform*, `http://www.eclipse.org/`.

[45] _____, *The Language Toolkit: An API for automated refactorings in Eclipse-based IDEs*, `http://www.eclipse.org/articles/Article-LTK/ltk.html`.

[46] F. Del Frate, P. Garg, A.P. Mathur, and A. Pasquini, *On the correlation between code coverage and software reliability*, ISSRE, 1995.

[47] Svend Frølund, *Coordinating distributed objects - an actor-based approach to synchronization*, MIT Press, 1996.

[48] Svend Frølund and Gul Agha, *A language framework for multi-object coordination*, ECOOP, 1993.

[49] Milos Gligoric, Tihomir Gvero, Vilas Jagannath, Sarfraz Khurshid, Viktor Kuncak, and Darko Marinov, *Test generation through programming in UDITA*, ICSE, 2010.

[50] Milos Gligoric, Vilas Jagannath, Qingzhou Luo, and Darko Marinov, *Efficient mutation testing of multithreaded code*, Wiley STVR, 2012.

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

[52] Mary Jean Harrold, James A. Jones, Tongyu Li, Donglin Liang, and Ashish Gujarathi, *Regression test selection for Java software*, OOPSLA, 2001.

[53] Thomas A. Henzinger, Ranjit Jhala, Rupak Majumdar, and Marco A. A. Sanvido, *Extreme model checking*, Springer Verification: Theory and Practice, 2003.

[54] Gerard J. Holzmann, Rajeev Joshi, and Alex Groce, *Tackling large verification problems with the swarm tool*, SPIN, 2008.

[55] Hwa-You Hsu and Alessandro Orso, *MINTS: A general framework and tool for supporting test-suite minimization*, ICSE, 2009.

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

[57] Vilas Jagannath, Milos Gligoric, Dongyun Jin, Qingzhou Luo, Grigore Roşu, and Darko Marinov, *Improved multithreaded unit testing*, ESEC/FSE, 2011.

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

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

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

[61] Wei Jin, Alessandro Orso, and Tao Xie, *Automated behavioral regression testing*, ICST, 2010.

[62] James A. Jones and Mary Jean Harrold, *Test-suite reduction and prioritization for modified condition/decision coverage*, ICSM, 2001.

[63] Pallavi Joshi, Mayur Naik, Chang-Seo Park, and Koushik Sen, *CalFuzzer: An extensible active testing framework for concurrent programs*, CAV, 2009.

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

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

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

[67] Joydip Kundu and Jan Emily Cuny, *A scalable, visual interface for debugging with event-based behavioral abstraction*, FMPC, 1995.

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

[69] Steven Lauterburg, Ahmed Sobeih, Darko Marinov, and Mahesh Viswanathan, *Incremental state-space exploration for programs with dynamically allocated data*, ICSE, 2008.

[70] Choonghwan Lee, Feng Chen, and Grigore Roşu, *Mining parametric specifications*, ICSE, 2011.

[71] David Leon and Andy Podgurski, *A comparison of coverage-based and distribution-based techniques for filtering and prioritizing test cases*, ISSRE, 2003.

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

[73] ———, *Tool support for testing concurrent Java components*, IEEE TSE (2003).

[74] J.A. Makowsky and E.V. Rawe, *Incremental model checking for fixed point properties on decomposable structures*, MFCS, 1995.

[75] Christopher D. Manning, Prabhakar Raghavan, and Hinrich Schtze, *Introduction to information retrieval*, Cambridge University Press, 2008.

[76] Darko Marinov and Sarfraz Khurshid, *TestEra: A novel framework for testing Java programs*, ASE, 2001.

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

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

[79] Nachiappan Nagappan, Andreas Zeller, Thomas Zimmermann, Kim Herzig, and Brendan Murphy, *Change bursts as defect predictors*, ISSRE, 2010.

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

[81] Adrian Nistor, Qingzhou Luo, Michael Pradel, Thomas Gross, and Darko Marinov, *BALLERINA: Automatic generation and clustering of efficient random unit tests for multithreaded code*, ICSE, 2012.

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

[83] OW2 Consortium, *ASM Framework*, `http://asm.ow2.org/`.

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

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

[86] Suzette Person, Matthew B. Dwyer, Sebastian G. Elbaum, and Corina S. Pasareanu, *Differential symbolic execution*, FSE, 2008.

[87] William Pugh and Nathaniel Ayewah, *MultithreadedTC - A framework for testing concurrent Java applications*, `http://code.google.com/p/multithreadedtc/`.

[88] ———, *Unit testing concurrent software*, ASE, 2007.

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

[90] Mathias Ricken and Robert Cartwright, *ConcJUnit: Unit testing for concurrent programs*, PPPJ, 2009.

[91] Gregg Rothermel and Mary Jean Harrold, *A safe, efficient regression test selection technique*, ACM TOSEM (1997).

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

[93] David Saff and Michael D. Ernst, *An experimental evaluation of continuous testing during development*, ISSTA, 2004.

[94] Koushik Sen, Darko Marinov, and Gul Agha, *CUTE: A concolic unit testing engine for C*, ESEC/FSE, 2005.

[95] Adam M. Smith, Joshua Geiger, Gregory M. Kapfhammer, and Mary Lou Soffa, *Test suite reduction and prioritization with call trees*, ASE, 2007.

[96] Oleg Sokolsky and Scott A. Smolka, *Incremental model checking in the modal mu-calculus*, CAV, 1994.

[97] Spring Source and Groovy Community, *GROOVY-1890*, `http://jira.codehaus.org/browse/GROOVY-1890`.

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

[99] Kuo-Chung Tai and Richard H. Carver, *Use of sequencing constraints for specifying, testing, and debugging concurrent programs*, ICPADS, 1994.

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

[101] Nikolai Tillmann and Jonathan de Halleux, *Pex—White box test generation for .NET*, TAP, 2008.

[102] University of Nebraska Lincoln, *Software-artifact Infrastructure Repository*, `http://sir.unl.edu/portal/index.html`.

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

[104] Kristen R. Walcott, Mary Lou Soffa, Gregory M. Kapfhammer, and Robert S. Roos, *Time-aware test suite prioritization*, ISSTA, 2006.

[105] Guoqing (Harry) Xu and Atanas Rountev, *Regression test selection for AspectJ software*, ICSE, 2007.

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

[107] S.S. Yau and Z. Kishimoto, *A method for revalidating modified programs in the maintenance phase*, COMPSAC, 1987.

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

[109] Andreas Zeller, *Yesterday, my program worked. today, it does not. why?*, ESEC/FSE, 1999.

[110] Lu Zhang, Shan-Shan Hou, Chao Guo, Tao Xie, and Hong Mei, *Time-aware test-case prioritization using integer linear programming*, ISSTA, 2009.