Program Analyses for Automatic and Precise Error Detection

Michael Pradel

Diss. ETH No. 20900

Program Analyses for Automatic and Precise Error Detection

Dissertation
submitted to
ETH Zurich
for the degree of
Doctor of Sciences

by

Michael Pradel
Diplom-Informatiker, Technische Universität Dresden
Diplôme d'Ingénieur, Ecole Centrale Paris
born March 10, 1983
citizen of Germany

accepted on the recommendation of
Prof. Dr. Thomas R. Gross, examiner
Prof. Dr. Jonathan Aldrich, co-examiner
Prof. Dr. Andreas Zeller, co-examiner

2012

Abstract

One of the largest challenges in software development is to ensure that the software is correct. Almost all software that is complex enough to accomplish a useful task contains programming errors. Unfortunately, developers must allocate their time to various activities and often, they do not have enough time for searching programming errors.

The goal of this dissertation is to support developers in finding programming errors despite a limited time budget. Therefore, we focus on program analyses with three properties. First, the analyses are *automatic*, that is, the only input required to analyze a program is the source code (or byte code) of the program itself. In particular, an automatic analysis does not rely on formal specifications or manually written test suites. Second, the analyses are *precise*, that is, they report warnings that are guaranteed to point to programming errors or that have a high chance of pointing to programming errors, instead of false positives. Third, the analyses can be applied to real-world software with *low human and computational effort*, that is, they provide developers a push button approach for existing code.

This dissertation argues that automatic program analysis allows for precisely detecting errors with little effort. The key idea is to leverage programs as their own oracles, for example, by leveraging a program as an executable specification for itself or by checking a program against properties inferred from the program itself. To support our thesis, we present five automatic and precise program analyses that effectively and efficiently detect programming errors. The analyses presented in this dissertation consider different kinds of errors (for example, incorrect API usages and thread safety violations), different kinds of programs (sequential and concurrent), and leverage different analysis techniques (static and dynamic). We evaluate our approach with mature and well-tested Java and C programs and show that it reveals errors automatically, precisely, and with low effort.

Zusammenfassung

Eine der grössten Herausforderungen der Softwareentwicklung ist es, die Korrektheit eines Programmes sicherzustellen. Die inhärente Komplexität jeder anspruchsvollen Software führt mit sehr hoher Wahrscheinlichkeit zu Programmierfehlern. In der ihnen zur Verfügung stehenden Zeit haben Entwickler allerdings selten die Möglichkeit, eine gründliche Fehlersuche zu realisieren.

Das Ziel dieser Dissertation ist es, Softwareentwicklern Möglichkeiten aufzuzeigen, Programmierfehler trotz eines knappen Zeitbudgets zu finden. Um dieses Ziel zu erreichen, präsentieren wir Programmanalysen mit den drei folgenden Eigenschaften: 1) Die Analysen sind automatisch. Die einzige Eingabe, die eine Analyse benötigt, ist der Quelltext des zu analysierenden Programmes. Insbesondere werden weder ausführbare Tests noch eine formale Spezifikation benötigt. 2) Die Analysen sind präzise. Die einzige Ausgabe, die eine Analyse liefert, sind Warnungen, welche mit Sicherheit oder mit sehr hoher Wahrscheinlichkeit auf tatsächliche Programmierfehler hinweisen. Insbesondere produziert eine präzise Analyse wenige oder gar keine falsch-positive Warnungen. 3) Die Analysen sind nicht aufwendig. Sie benötigen nur einen geringen Rechenaufwand und Softwareentwickler können sie mit wenig Arbeitsaufwand nutzen.

Die vorliegende Arbeit vertritt die These, dass automatische und präzise Programmanalyse helfen kann, mit geringem Aufwand Programmierfehler zu finden. Die Kernidee hierbei ist, Programme als ihre eigenen Testorakel zu nutzen. So verwenden wir beispielsweise ein Programm als ausführbare Spezifikation für sich selbst, oder überprüfen, ob ein Programm Spezifikationen einhält, welche vom Programm selbst abgeleitet wurden. Wir stellen fünf Programmanalysen vor, welche wirksam und effizient Fehler aufdecken. Diese Programmanalysen betrachten verschiedene Arten von Fehlern (z.B. inkorrekte API-Benutzungen oder Verletzungen von Thread Safety), verschiedene Arten von Programmen (sequentiell und nebenläufig), und verwenden diverse Analysetechniken (statisch und dynamisch). Zur Evaluierung unseres Ansatzes wenden wir die Analysen auf ausgereifte und praxiserprobte Java- und C-Programme an. Unsere Ergebnisse zeigen, dass die Analysen Programmierfehler automatisch, präzise und ohne grossen Aufwand finden.

Acknowledgments

The work described in this dissertation took about four years from conception to completion. This period of time included countless cycles of exploration, inquiry, meditation, enlightenment, doubt, confusion, uncertainty, and perseverance, which I could not have faced without the help of many others.

First of all, I want to thank my adviser, Thomas R. Gross. Mentioning your adviser first is common, but this is not the reason why I follow this tradition. The reason is that I am deeply grateful for the various pieces of advice on doing and presenting research. Thomas has given me the freedom to create and follow my own research agenda and, at the same time, has provided guidance whenever needed. This unique combination has allowed me to learn from his vast experience while becoming an independent researcher.

Special thanks to the external members of my PhD commitee, Jonathan Aldrich and Andreas Zeller, for committing to this duty and for their valuable feedback.

Thanks to the many people at ETH who made my PhD journey an unforgettable trip. To Zoltán Majó for sharing the greenest office in the institute with me and for the many spontaneous discussions on research, life, and Switzerland. To the "older" group members, Christoph Angerer, Stephanie Balzer, Nicholas D. Matsakis, Albert Noll, Mathias Payer, Susanne Cech Previtali, Florian Schneider, Yang Su, and Oliver Trachsel, for sharing their insights on research, teaching, and dealing with the challenges of being a PhD student. To the "younger" group members, Mihai Cuibus, Stefan Schmid, Luca Della Toffola, Faheem Ullah, and Fabio Zünd, for accompanying me on my way through the doctoral studies. Our group lunches have been a welcome distraction from the daily research grind and I will certainly remember the many enjoyable tea/coffee session in BQM.

I want to thank my external collaborators, Ciera Jaspan and Jonathan Aldrich from CMU, as well as Adrian Nistor, Qingzhou Luo, and Darko Marinov from UIUC. I am grateful for the fruitful collaborations, which allowed me to discover new perspectives on my research. During my doctoral studies, I had to privilege to supervise several bachelor and master students, whom I owe a dept of gratitude for their excellent work: Philipp Bichsel, Jérémie Bresson, Claudio Corrodi, Severin Heiniger, Sebastian Grössl, Christine Zeller, and Pascal Zimmermann.

Many thanks to the those who gave feedback on paper drafts and talks: Eric Bodden, Mark Gabel, David Lo, Patrick Meredith, Markus Püschel, Grigore Rosu, Friedrich Steimann, Marco Zimmerling, and the anonymous reviewers.

Thanks to my earlier research advisers, Uwe Assmann, Jakob Henriksson, and Martin Odersky, who have motivated me to pursue a research career and who have helped me to take my first steps in the academic world.

To implement our ideas, we build upon tools provided by others. I want to thank Michael D. Ernst and Carlos Pacheco for providing Randoop, William W. Cohen, Pradeep Ravikumar, Stephen Fienberg, and Kathryn Rivard for prodiving Second-String, and the Soot community, in particular, Eric Bodden and Patrick Lam.

Finally, I want to thank my family and friends, in particular, my wife Antje and my son Paul, for their unconditional love and support.

Short Contents

Contents

Introduction 1

1.1 Motivation

Today's society strongly depends on software. Hospitals have servers to manage patient records and software-controlled devices are crucial to keep people alive. Various industries rely on software for producing, managing, and selling their products. Mobility of people and goods depends on software because trains, airplanes, and traffic control systems are software-driven. The internet, which by now is hard to imagine to live without, is a tremendous software system built from various client and server programs. Finally, most people rely on software in their every-day life, for example, when communicating with each other via email or phone.

Given the enormous dependence of our society on software, one could naively expect most software to be correct and reliable. Unfortunately, it is not: Most software has bugs.[1] For average industry code, the number of bugs per 1,000 lines of code has been estimated to range between 0.5 and 25 for delivered software [107]. Even after years of deployment, popular pieces of software still contain unnoticed bugs that pop up at some point in time. For example, studies of the Linux kernel show that the average bug remains in the kernel for a surprisingly long period of 1.5 to 1.8 years [29, 118].

A single bug can cause serious harm, even if it has been subsisting for a long time without doing so. The Ariane 5 space rocket was destroyed 40 seconds after takeoff due to an outside-expected-range value in a piece of software successfully used for Ariane 4 [93]. A race condition in the Unix-based XA/21 energy management system caused the Northeast blackout, a power outage that affected large parts of the U.S. and Canada in 2003 [123]. Zhivich and Cunningham discuss more examples of software bugs that have caused huge economic loses and even have killed people [177]. All these examples illustrate that identifying and fixing bugs is a crucial part of software development.

There are various techniques to increase the correctness and reliability of software. Among the techniques that apply directly to the software (rather than, for example, to the software development process), verification and testing are the most common. Verification provides a mathematical proof that a program fulfills a formal specification and therefore shows the absence of a particular kind of bug. In contrast, testing

[1]We define the colloquial term "bug" in Section 1.2.

aims at discovering bugs by exploring usage scenarios of a program. Both verification and testing are used at various levels of granularity, ranging from small code fragments to entire software systems.

Unfortunately, neither verification nor testing can guarantee bug-free software. A verified program may behave incorrectly because verification relies on a formal specification that may be incorrect or that may not cover a particular kind of misbehavior. Testing relies on a test suite that exercises a program. Similar to a formal specification, such test suites can contain bugs. Moreover, test suites typically cover only a strict subset of all possible usage scenarios of a program because exhaustive testing is infeasible for complex systems. As a result, even well-tested programs may have bugs.

Since there is no silver bullet to ensure the correctness and reliability of software [17], developers apply techniques that bring them closer to this goal. Given tight deadlines, an important criterion in choosing these techniques are their costs. Both verification and testing require developers to invest significant effort: Developers must create artifacts in addition to the program (formal specifications and test suites, respectively), maintain these additional artifacts, and adapt them to changes in the requirements and in the program.

The need to increase correctness and reliability, combined with the high costs of existing techniques for this purpose raises the question: *Can we provide developers with low cost techniques that increase the correctness and reliability of software?* This thesis addresses this question and proposes a promising solution that helps developers find bugs with little effort.

1.2 Terminology

We use the term *programming error*, or *bug*, for a part of a program that developers should change because the current implementation unintentionally leads to unexpected behavior, reduces performance, or makes maintaining the program unnecessarily difficult.

For example, we consider the following problems to be programming errors:

- The program's output is 23 but should be 42.
- The program crashes unexpectedly.
- The program performs an unnecessary computation that leads to a longer running time.
- The meaning of a method or variable is hard to understand because their names are badly chosen.

Our terminology is in line with the IEEE Standard Glossary of Software Engineering Terminology [3]. The glossary defines an "error" as an incorrect step, process, or data definition in a computer program, and mentions "bug" and "fault" as common synonyms.

1.3 Automatic and Precise Bug Detection

This thesis argues that *automatic program analysis allows for precisely detecting programming errors with little effort*. We present a class of program analyses that have three properties:

- *Automatic*. The analysis can analyze existing programs as they are, without any manual preparation. In particular, the only input to an automatic analysis is the program's source code (or byte code), without relying on manually written formal specifications or tests.
- *Precise*. The analysis reports warnings that are guaranteed to point to programming errors or that have a high chance of pointing to programming errors. We call warnings that point to programming errors *true positives*, and call other, spurious warnings *false positives*. If the analysis does not guarantee that all warnings are true positives, then it provides a "knob" that allows a user to control the trade off between precision and finding more bugs.
- *Little effort*. The analysis can be applied to real-world software with low human and computational effort.

A program analysis with these three properties is an attractive approach for increasing the correctness and reliability of real-world software. Experience from applying analysis tools to industrial code underlines the importance of the three properties [13]: A successful analysis must be easily applicable to existing programs and, at the same time, report bugs with high precision.

To evaluate our hypothesis that automatic program analysis allows for precisely detecting programming errors with little effort, we develop and apply five program analyses. The analyses fall into two broad categories:

- *Use a program as an executable specification for itself*. This approach tests a program by leveraging the program as an executable specification for itself. The key idea is to use classes of the program in two ways that are supposed to expose the same behavior. If the behavior of one usage diverges in a significant way from the behavior of the other usage, the analysis raises a warning. We present two techniques of this kind, which discover thread safety violations and classes that are unsafe substitutes for their superclasses, respectively.
- *Infer probabilistic usage patterns and search for anomalies*. This approach involves two steps. At first, an analysis infers patterns from existing usages of a method or a set of methods. Based on the assumption that most of the analyzed usages are correct, these patterns are considered to be probabilistic specifications. Then, the analysis checks for violations of common patterns that may be due to a programming error. We describe three techniques of this kind, which reveal API protocol violations, incorrectly ordered method arguments, and method arguments that have a compatible but unexpected type, respectively.

The program analyses we advocate in this thesis are incomplete, that is, they can show the presence of bugs but cannot guarantee their absence. Experience with static checkers, such as FindBugs [75], shows that incomplete approaches can be very effective in practice [11, 13]. Therefore, we deliberately accept incompleteness and focus on approaches that are automatic, precise, and that can be used with little effort.

1.4 Contents and Contributions

The main contribution of this work is to demonstrate that automatic program analysis allows for precisely detecting programming errors with little effort. We support this thesis with five independent program analyses that consider different kinds of programming errors, address different kinds of programs (sequential and concurrent), and leverage different kinds of analyses (static and dynamic). In the following, we summarize the key insights of each analysis and how it contributes to the state of the art. Chapter 7 discusses related work and compares our work to existing approaches.

Chapter 2: Thread Safety Violations

Concurrent, object-oriented programs often use thread-safe library classes. Chapter 2 presents an automatic testing technique that reveals concurrency bugs in supposedly thread-safe classes. The key idea is to generate tests in which multiple threads call methods on a shared instance of the tested class. If a concurrent test exhibits an exception or a deadlock that cannot be triggered in any linearized execution of the test, the analysis reports a thread safety violation. The analysis finds concurrency bugs in popular Java libraries, including two previously unknown bugs in the Java standard library.

Contributions

This work contributes by increasing automation and precision compared to the closest existing approach [18]. The existing approach generates tests based on programmer-provided method calls to the class under test. Instead, our approach generates tests in a fully automatic way that requires as only input the class under test. Our analysis reports only true positives because it focuses on exceptions and deadlocks that happen when using a class concurrently but that cannot happen when using it sequentially. Instead, the closest existing approach classifies 426 of 1,800 test executions as failing, which – after manual inspection – contain seven bugs [18].

Chapter 2 shares material with the PLDI'12 paper "Fully Automatic and Precise Detection of Thread Safety Violations" [127].

Chapter 3: Unsafe Substitutes

Languages with inheritance and polymorphism assume that a subclass instance can substitute a superclass instance without causing behavioral differences for clients of the superclass. However, programmers may accidentally create subclasses that are semantically incompatible with their superclasses. Such subclasses lead to bugs because a programmer may assign a subclass instance to a superclass reference. Chapter 3 presents an automatic testing technique to reveal subclasses that cannot safely substitute their superclasses. The key idea is to generate generic tests that analyze the behavior of both the subclass and its superclass. If using the subclass leads to a behavior that cannot occur with the superclass, the analysis reports a warning. We find a high percentage of widely used Java classes, including classes from JBoss, Eclipse, and Apache Commons Collections, to be unsafe substitutes for their superclasses: 30% of these classes lead to crashes and even more of them have other behavioral differences.

Contributions

To the best of our knowledge, this analysis is the first to address unsafe subclasses with an automatic analysis. Previous work on ensuring substitutability has focused on verifying this property [90, 8, 95, 41, 137]. A verification approach requires programmers to formally specify the behavior of superclasses and subclasses. In contrast, our approach does not rely on formal specifications and therefore increases automation.

Chapter 3 shares material with the ICSE'13 paper "Automatic Testing of Sequential and Concurrent Substitutability" [129].

Chapter 4: API Protocol Violations

Programmers using an API often must follow protocols that prescribe the order of method calls. Such protocols can involve multiple interacting objects, such as a collection and an iterator over the collection. Chapter 4 presents an approach that combines test generation, protocol mining, and runtime verification into a fully automatic dynamic analysis to find unsafe API usages that violate a protocol. The approach leverages generated tests in two ways: Passing tests drive the program during protocol mining, and failing test executions are checked against inferred protocols. The output are warnings that show with concrete test cases how the program violates commonly accepted protocols. The analysis reports no false positives and 54 true positives in ten well-tested Java programs.

Contributions

The approach is the first that finds bugs based on inferred specifications while not reporting false positives. The key idea is to focus on violations of inferred protocols that cause an exception, that is, certainly undesired behavior. Moreover, the analysis is the first to automate dynamic specification mining and runtime verification with generated tests.

Chapter 4 shares material with the ICSE'12 paper "Leveraging Test Generation and Specification Mining for Automated Bug Detection without False Positives" [128], the ICSM'10 paper "A Framework for the Evaluation of Specification Miners Based on Finite State Machines" [124], and the ASE'09 paper "Automatic Generation of Object Usage Specifications from Large Method Traces" [125].

Chapter 5: Incorrectly Ordered, Equally Typed Arguments

In statically typed programming languages, the compiler ensures that method arguments are passed in the expected order by checking the type of each argument. However, calls to methods with multiple equally typed parameters slip through this check. The uncertainty about the correct order of equally typed arguments can cause various problems, for example, if a programmer accidentally reverses two arguments. Chapter 5 presents a static analysis that detects such problems without any input except for the source code of a program. The analysis leverages the observation that programmer-given identifier names convey information about the semantics of arguments, which can be used to assign equally typed arguments to their expected position. We evaluate the approach with a large corpus of Java programs and show that the analysis finds relevant anomalies with a precision of 76%.

Contributions

This work contributes by introducing the notion of anomalies of equally typed arguments, a kind of problem not considered by any previous work on bug finding. The closest existing approach is to prevent incorrectly ordered arguments with named parameters, a language construct offered, for example, by Scala and Smalltalk. Named parameters impose additional annotation overhead to programmers and many popular languages, such as Java, do not provide them. In contrast, our approach searches for bugs in a fully automatic way.

Chapter 5 shares material with the ISSTA'11 paper "Detecting Anomalies in the Order of Equally-typed Method Arguments" [126].

Chapter 6: Brittle Parameter Types

To avoid receiving incorrect arguments, a method specifies the expected type of each formal parameter. However, some parameter types are too general and have subtypes that the method does not expect as actual argument types. For example, this may happen if there is no common supertype that precisely describes all expected types. As a result of such *brittle parameter types*, a caller may accidentally pass arguments unexpected by the callee without any warnings from the type system. Chapter 6 presents a static analysis to find brittle parameter types and unexpected arguments given to them. First, the analysis infers from callers of a method the types that arguments commonly have. Then, the analysis reports potentially unexpected arguments that stand out by having an unusual type. We apply the approach to 21 real-world Java programs that use the Swing API, an API providing various methods with brittle parameters. The analysis reveals previously unknown bugs and code smells and has a precision of 47%.

Contributions

The analysis contributes by identifying brittle parameters as a source of programming errors and by presenting the first bug finding technique to address this problem.

Chapter 6 shares material with the ISSTA'12 paper "Static Detection of Brittle Parameter Typing" [130].

1.5 Potential Impact

We hope and expect this thesis to have immediate practical impact as well as long-term impact on the research community. The thesis presents techniques that are ready to be used in practice. We present prototype implementations of all analyses presented in this thesis and evaluate them with large, real-world programs. Our results show that the analyses detect relevant problems in an automatic and precise way that requires little effort. During our evaluation, the analyses found a total of over 200 programming errors. We reported bugs to the developers of various open source projects, including Sun's Java standard library, Eclipse, Apache Commons Collections, and AspectJ. At the time of this writing, we are collaborating with a major software company that is interested in adopting some of our techniques.

The broader impact of this thesis is to show how powerful automatic approaches are for finding bugs. Society depends more and more on software. At the same time, no single solution to avoid software bugs is to be expected in the foreseeable future. Therefore, the need for low-cost techniques to improve the correctness and reliability of software will increase. Our work addresses this need in a promising way. We expect more research on automatic program analyses for finding bugs and hope that future work benefits from the results of this thesis.

1.6 Further Resources

To facilitate others to reproduce our results and to compare their work with ours, Appendices A and B list details about the programs we analyze and the warnings reported by our approaches, respectively. Furthermore, we provide a web site with additional material, such as source code of our prototype implementations:

`http://mp.binaervarianz.de/dissertation/`

The analysis presented in Chapter 2 is available at:

`http://www.thread-safe.org`

2 Thread Safety Violations

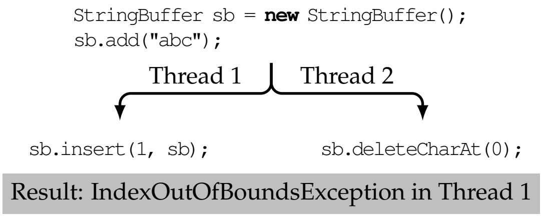
Writing correct concurrent programs is hard. Since developers are used to sequential reasoning, the parallelism and non-determinism of concurrent programs makes these programs hard to write and understand. The problem is compounded because developers do not have enough tool support for testing concurrent programs [60]. Testing techniques for concurrency have not yet reached the sophistication of techniques for sequential programs.

Existing approaches to find concurrency bugs are either not automatic, not precise, or neither automatic nor precise. Existing dynamic analyses are not automatic because they rely on tests that exercise the software under test. Writing tests is often neglected due to time constraints. Furthermore, writing effective concurrent tests is difficult because they should lead the program to sharing state between concurrently executing threads and should trigger many different execution paths. Another reason for not being automatic is that many existing static and dynamic analyses rely on explicit, formal specifications, which must be developed in addition to the program itself. Unfortunately, few programs provide such specifications. Many existing approaches are not precise because, in addition to true positive, they report false positives.

This chapter addresses concurrency bugs in thread-safe classes. A class is *thread-safe* if it “behaves correctly when accessed from multiple threads, regardless of the scheduling or interleaving of the execution of those threads by the runtime environment, and with no additional synchronization or coordination on the part of the calling code” [61]. We say that a class is *thread-unsafe* otherwise. Thread safety is a widely used correctness guarantee, and thread-safe classes serve as building blocks for creating concurrent programs. Many libraries offer thread-safe classes and the correctness of programs using them relies on the correctness of the thread safety classes.

Unfortunately, ensuring that a class is thread-safe is non-trivial, and even mature and well-tested classes that are supposedly thread-safe may turn out to be thread-unsafe in some usages. As a motivating example, consider a previously unknown bug that our approach detects in the `StringBuffer` class of Sun’s Java standard library in versions 1.6 and 1.7.[1] `StringBuffer` is documented as thread-safe and synchronizes accesses to its internal data by locking on the `StringBuffer` instance.

[1]We reported the problem and the developers acknowledged it as a bug. See entry 7100996 in Sun’s bug database.



```
StringBuffer sb = new StringBuffer();
sb.add("abc");
```

```
sb.insert(1, sb);            sb.deleteCharAt(0);
```

(a) Concurrent usage that exposes a thread safety violation.

```
class StringBuffer {
  StringBuffer(String s) {
    // initialize with the given String
  }
  synchronized void deleteCharAt(int index) {
    // modify while holding the lock
  }
  void insert(int dstOffset, CharSequence s) {
    int l = s.length();
    // BUG: l may change
    this.insert(dstOffset, s, 0, l);
  }
  synchronized void insert(int dstOffset,
      CharSequence s, int start, int end) {
    // modify while holding the lock
  }
}
```

(b) Supposedly thread-safe class.

Figure 2.1: A thread safety violation in `StringBuffer`.

For example, Figure 2.1a shows a multi-threaded usage of `StringBuffer`, where a single-threaded sequence of calls is followed by multiple concurrent threads that use a shared variable `sb`.[2] This multi-threaded usage of `StringBuffer` is legal and should not cause any problems. However, executing the example leads to an exception because `insert()` retrieves the length of `s`, a parameter of type `CharSequence`, before acquiring the lock. The documentation of `StringBuffer` states: "This class synchronizes only on the string buffer performing the operation, not on the source." This behavior is fatal if the passed source is the `StringBuffer` itself, a case the developers of the class apparently forgot to consider (Figure 2.1b). In this case, retrieving the size of the `StringBuffer` is non-atomic with the consecutive update, leading to a potential boundary error caused by an interleaved update.

[2]We use a graphical notation for representing code that runs concurrently. The example in Figure 2.1a is equivalent to Java code that creates two threads after the call to `add()` and that starts them with `Thread.start()`.

In this chapter, we present an automatic and precise bug detection approach to find thread safety violations. The approach is automatic because it requires a single input: the class under test (CUT), possibly accompanied by other classes on which the CUT depends. The approach is precise because it produces a single output: true positive reports about concurrency bugs in the CUT. We implement the approach and apply it to six popular code bases, including the Java standard library and the Apache Commons Database Connection Pools (DBCP) library. The analysis detects previously unknown bugs, for example, the problem illustrated in Figure 2.1. To find these violations, the analysis requires minimal human effort and an acceptably low computational effort.

The following two sections outline our approach and illustrate it with a detailed example. Sections 2.3 and 2.4 explain the two contributions that enable the approach, followed by a description of our prototype implementation in Section 2.5. Section 2.6 evaluates the approach. Sections 2.7 argues why the approach supports the thesis of this work, and finally, Section 2.8 outlines directions of future work.

2.1 Overview of the Approach

This section explains the key ideas of a dynamic analysis to detect concurrency bugs in thread-safe classes. There are three requirements for detecting a thread safety bug with a dynamic analysis:

1. A test that drives an instance of the CUT to a state that can expose the bug.
2. An execution environment that exposes the bug when executing the test.
3. An oracle that recognizes the execution as erroneous.

Various approaches addressing the second requirement have been proposed [42, 140, 110, 35, 19]. This work focuses on the first and the third requirement, and we contribute two techniques that address these requirements.

The first contribution is a test generation technique that creates input to exercise the methods of a CUT from multiple threads. Each generated test consists of a sequential part, which instantiates the CUT, and a concurrent part, in which multiple

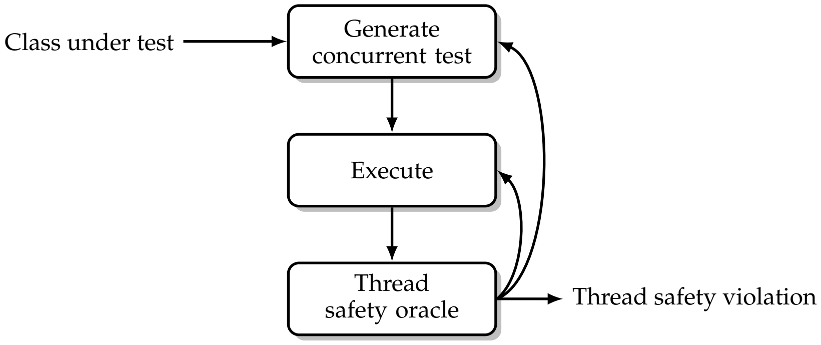


Figure 2.2: Overview of the approach.

threads call supposedly thread-safe methods of the CUT instance. The technique enhances existing techniques that generate sequential tests [36, 117] by adapting them to a concurrent setting. Simply running generated sequential tests in parallel is very unlikely to result in shared state and to expose concurrency bugs. Instead, our technique generates tests that share a single CUT instance, called the *object under test* (OUT), between multiple threads and that expose concurrency bugs by using the OUT concurrently.

Some existing bug finding techniques use a manually written test harness that, given a set of calls with concrete arguments, randomly selects and executes calls [67, 18]. Our test generation technique advances upon this approach in two ways. First, it relieves developers from providing calls and from finding appropriate arguments for these calls. The test generator creates arguments by instantiating classes and by calling methods that return objects of the required types. Second, and even more important, an automatic approach produces more diverse tests than manually written calls because it tries to combine many different methods, arguments, and receiver objects. As a result, generated tests include usage scenarios that a human may not come up with. For example, the test in Figure 2.1a triggers a bug by passing a `StringBuffer` to itself, a situation that apparently remained untested for several years.

The second contribution is a test oracle, called the *thread safety oracle*, that determines whether the execution of a concurrent test exposes a thread safety violation. The oracle classifies a concurrent execution as erroneous if the execution leads to an exception or to a deadlock *and* if this exception or deadlock cannot be triggered by any linearization of the calls in the concurrent execution. A *linearization* maps all calls of a concurrent test into a single thread while preserving the order of calls made by individual threads. The thread safety oracle is generic, automatic, and precise. It is generic because it can detect different kinds of concurrency bugs, including data races, atomicity violations, and deadlocks, given that the bug eventually causes an exception or a deadlock. A study of 105 real-world concurrency bugs found that 62% of all bugs lead to a crash or a deadlock [100], suggesting that the thread safety oracle addresses a significant problem. The oracle is automatic because it does not require any explicit specification of the CUT and instead leverages generic and implicit indicators of incorrectness. Finally, the oracle is precise because it guarantees that each reported problem is a bug, as exceptions and deadlocks are certainly undesired behavior.

A main assumption of the oracle is that sequential executions are deterministic. If this assumption was false, the oracle could not guarantee precision. The reason is that the oracle might non-deterministically miss an exception or a deadlock in a linearization and might incorrectly conclude that the exception or the deadlock occurs only concurrently. In practice, we find that most thread-safe classes are sequentially deterministic. For classes that are sequentially non-deterministic, our analysis can be combined with a runtime environment that ensures deterministic sequential execution [122].

The thread safety oracle relates to seminal work on linearizability as a correctness criterion [73] and its recent adaption to object-oriented programs in Line-Up [18]. In contrast to Line-Up, our oracle is more effective and more efficient. It is more effective because each test execution that fails according to the oracle is a true positive. In contrast, the Line-Up oracle classifies 426 of 1,800 test executions as failing, which—after manual inspection—contain seven bugs [18]. This high violations-to-bugs ratio is due to benign linearizability violations and to multiple violations caused by the same bug. Our oracle is more efficient because it only runs linearizations of a concurrent test if the concurrent test leads to an exception or a deadlock. Instead, Line-Up explores all linearizations before running a concurrent test.[3]

Our analysis combines these two contributions into a system that iteratively performs three steps (Figure 2.2), which correspond to the three requirements. At first, the test generator creates a test that sets up the OUT and that calls methods of the OUT from multiple threads. Next, we execute the generated test. Since the class is supposedly thread-safe, it should synchronize concurrent calls on the OUT as needed. That is, the methods should "behave as if they occur in some serial order that is consistent with the order of the method calls made by each of the individual threads involved" [4]. The third step of our analysis checks whether the OUT behaves as expected. If the oracle finds that the concurrent execution shows behavior not possible with any linearization of the calls in the test, then it reports a thread safety violation and terminates. Otherwise, the analysis goes back to the first step and continues until a stopping criterion, such as a timeout or a maximum number of generated tests, is reached.

2.2 Detailed Example

Before explaining the details of our approach, we walk through an example. Suppose a developer wants to test the `StringBuffer` class, which is documented to be thread-safe. All the developer must do is to provide the source code or byte code of the class.

2.2.1 Generating a Concurrent Test

As a first step, the analysis generates a concurrent test that exercises a `String-Buffer` instance. To get such an instance, the test generator randomly chooses among the public constructors of `StringBuffer`. Suppose the generator chooses the default constructor:

```
StringBuffer sb = new StringBuffer();
```

Whenever the test generator adds a constructor call or a method call to a test, it executes the test to check whether the added call leads to an uncaught exception or to a deadlock. If it does, then extending the test cannot explore further behavior because the test crashes when the added call is executed. Executing the default constructor of `StringBuffer` succeeds in this case, so the test generator continues to extend the test.

After instantiating the `StringBuffer`, the test generator calls methods on `sb` to bring the object in a state that may allow us to trigger a bug. For this purpose, the test generator randomly chooses a public and concrete method of `StringBuffer`, for example, `append(String)`. To call this method, we require a `String` argument. The test generator randomly chooses among the following three options:

- Use a `String` variable from the so far generated test. This option is not possible for the example because the test does not have any `String` variables.
- Call a method that returns a `String`.

[3] A direct comparison of our results with those of Line-Up is not possible because the details about the classes under test [18], as well as the specific bugs found, are no longer available [20].

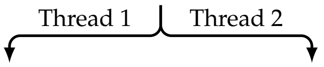
- Use a randomly generated value.

Suppose the test generator selects the last option and extends the test as follows:

```
StringBuffer sb = new StringBuffer();
sb.append("abc");
```

The test generator executes the so far constructed test and, again, finds that it succeeds. The goal is to generate a concurrent test that exercises `sb` from multiple concurrent threads. To achieve this goal, the test generator spawns two threads in the test and adds to each thread calls with `sb` as the receiver. For the first thread, the test generator chooses to call `insert(int,CharSequence)`. The generator requires an `int` argument and a `CharSequence` argument and chooses them based on the three above options. Suppose the generator extends the test as follows: [4]

```
StringBuffer sb = new StringBuffer();
sb.append("abc");
```

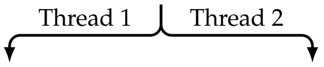


```
sb.insert(-5, sb);
```

To avoid adding calls that certainly lead to a crash of the test, the generator checks whether executing the three calls in a single thread of execution succeeds (that is, as if `sb.insert(-5, sb)` was added before forking threads). The execution raises an `IndexOutOfBoundsException` because `-5` is an invalid index. Instead of this call, the test generator tries to find other arguments that do not crash the test. Suppose it generates a call to `sb.insert(1, sb)`, which does not raise an exception.

Next, the test generator adds a call to the second thread by randomly choosing a method, selecting arguments for it, and checking whether adding the call fails the test. After this step, we obtain the following, concurrent test:

```
StringBuffer sb = new StringBuffer();
sb.append("abc");
```



```
sb.insert(1, sb);            sb.deleteCharAt(1);
```

2.2.2 Executing the Test

The second step of our analysis is to execute the generated test. Different interleavings of the concurrent threads may result in different behavior. One possible behavior, which we suppose to happen, is that `sb.insert(1, sb)` raises an `IndexOutOfBoundsException`. That is, executing the generated concurrent test leads to a crash.

[4]For presentation purposes, we use a graphical notation for creating and starting threads. Our implementation creates tests that use Java's standard thread facilities, for example, `Thread.start()`.

2.2.3 Thread Safety Oracle

Does the exception thrown by the concurrent test indicate an error? The third step of our analysis, the thread safety oracle, answers this question by leveraging the definition of thread safety. For a thread-safe class, the behavior obtained when executing method calls concurrently must be the same as the behavior obtained in a linearization at the granularity of method calls. For our example, there are two linearizations:

```
StringBuffer sb = new StringBuffer();
sb.append("abc");
sb.insert(1, sb);
sb.deleteCharAt(1);
```

and

```
StringBuffer sb = new StringBuffer();
sb.append("abc");
sb.deleteCharAt(1);
sb.insert(1, sb);
```

Executing neither of the two linearizations leads to an exception, that is, the crash occurs only when using `StringBuffer` concurrently. This behavior exposes a thread safety violation and our analysis reports the problem to the developer.

The example illustrates how using an automatic and precise analysis helps developers to find thread safety bugs with little effort. In the following sections, we explain the approach in more detail.

2.3 Generating Concurrent Tests

This section presents a technique to generate concurrent tests that exercise a CUT from multiple threads. The input to the test generator is a class C and, optionally, a set $\mathcal{A}$ of auxiliary classes that C depends on. The generator creates *call sequences*. Each call c_i in a call sequence $(c_1, \ldots, c_n)$ consists of a method signature, a possibly empty list of input variables, and an optional output variable. The input variables of a call represent arguments passed to the method. For an instance call, the first input variable is the receiver of the call. The output variable of a call represents its return value. We model constructor invocations as calls where the output variable represents the new object. Similarly, we model field accesses as calls where the underlying object is the only input variable and where the output variable represents the field value. We require all call sequences to be *well-defined*, that is, each input variable of a call c_j is the output variable of a call c_i with $i < j$, or in other words, each variable used in a call is defined by a prior call.

A *test* consists of a prefix and a set of suffixes. The *prefix* is a call sequence supposed to be executed in a single thread. The prefix instantiates the CUT and calls methods on the OUT to "grow" the object, that is, to bring it into a state that may allow the suffixes to trigger a bug. A *suffix* is a call sequence supposed to be executed concurrently with other suffixes after executing the prefix. All suffixes share the output variables of the prefix and can use them as input variables for suffix calls. In particular, all suffixes share the OUT created in the prefix. While our general approach is independent of the number of suffixes, we focus on two suffixes per test. That is, a test is a triple (p, s_1, s_2), where p is the prefix and s_1, s_2 are suffixes. The rationale for this choice is that most real-world concurrency bugs can be reproduced with at most two threads [100].

The test generated in Section 2.2 is a simple example for a test. The prefix contains a call to a constructor of `StringBuffer`, which returns the OUT `sb`, and a call on the OUT. Each of the two suffixes calls a single method using the shared OUT `sb` as the receiver. In practice, effective concurrent tests are not always that simple. Calling a method often requires providing arguments of a particular type, which in turn may require calling other methods. Furthermore, triggering a bug often requires bringing the OUT into a particular state, for example, by calling setter methods.

2.3.1 Tasks

We divide the generation of a test into tasks that build and extend call sequences. Each task takes a call sequence $s_{in} = (c_1, \ldots, c_i)$ and returns a new sequence $s_{out} = (c_1, \ldots, c_i, c_j, \ldots, c_n)$ that appends $n - j + 1$ additional calls to s_{in}. The additional calls can use output variables of previous calls as their input variables. We use three kinds of tasks:

- $instantiateTask$, which appends calls to create the OUT,
- $callTask$, which appends a call to the OUT, and
- $argumentTask$, which makes an argument of a particular type available.

A task succeeds if it extends s_{in} with additional calls in such a way that s_{out} executes in a single thread without throwing an uncaught exception. This requirement adapts an idea from sequential test generation [117], namely to use the result of executing call sequences for selecting which sequences to extend further. Generated call sequences that result in an exception typically contain an illegal call, for example, a call that violates the precondition for calling the method. Although focusing on non-exceptional call sequences does not guarantee that each call is legal, it ensures that extending a sequence will eventually lead to executing more calls without reaching an obviously illegal state.

To successfully accomplish a task, the test generator extends s_{in} into *candidates* for s_{out} until a candidate is found that executes without an exception. If no candidate has fulfilled these conditions after creating a configurable number of candidates, the test generator gives up on this task and the task fails. For example, the $callTask$ may fail if the OUT is at a state that does not allow calling any of the CUT methods.

2.3.2 Test Generation Algorithm

Algorithm 1 describes how our analysis generates a concurrent test. There are three global variables, which maintain their values over multiple invocations of the algorithm: the set $\mathcal{P}$ of prefixes, the map M assigning a prefix to its set of suffixes, and the set $\mathcal{T}$ of already generated but not yet returned tests.

The algorithm has three main steps. At first, it creates a new prefix or chooses a previously created prefix (lines 6 to 20). Then, it creates a new suffix for the prefix (lines 21 to 26). Finally, the algorithm creates tests by combining the new suffix with each existing suffix (lines 27 to 29). To create prefixes and suffixes, the algorithm invokes tasks. The functions $randTake$ and $randRemove$ randomly select an element of a set with a uniform probability distribution. The $randRemove$ function also removes the selected element from the set.

Algorithm 1 Returns a concurrent test (p, s_1, s_2)

```
1: $\mathcal{P}$: set of prefixes                                ▷ global variables
2: $M$: maps a prefix to suffixes
3: $\mathcal{T}$: set of ready-to-use tests
4: if $|\mathcal{T}| > 0$ then
5:     return $randRemove(\mathcal{T})$
6: if $|\mathcal{P}| < maxPrefixes$ then                          ▷ get a prefix
7:     $p \leftarrow instantiateTask$(empty call sequence)
8:     if $p = failed$ then
9:         if $\mathcal{P} = \emptyset$ then
10:            $fail$("cannot instantiate CUT")
11:        else
12:            $p \leftarrow randTake(\mathcal{P})$
13:    else
14:        for $i \leftarrow 1, maxStateChangerTries$ do
15:            $p_{ext} \leftarrow callTask(p)$
16:            if $p_{ext} \neq failed$ then
17:                $p \leftarrow p_{ext}$
18:    $\mathcal{P} \leftarrow \mathcal{P} \cup \{p\}$
19: else
20:    $p \leftarrow randTake(\mathcal{P})$
21: $s_1 \leftarrow$ empty call sequence                          ▷ create a new suffix
22: for $i \leftarrow 1, maxCallTries$ do
23:    $s_{1,ext} \leftarrow callTask(s_1, p)$
24:    if $s_{1,ext} \neq failed$ then
25:        $s_1 \leftarrow s_{1,ext}$
26: $M(p) \leftarrow M(p) \cup \{s_1\}$
27: for all $s_2 \in M(p)$ do                  ▷ one test for each pair of suffixes
28:    $\mathcal{T} \leftarrow \mathcal{T} \cup \{(p, s_1, s_2)\}$
29: return $randRemove(\mathcal{T})$
```

Creating Prefixes During the first step (lines 6 to 20), the algorithm creates a new prefix unless $maxPrefixes$ (discussed in Section 2.5) have already been created. To create a new prefix, the algorithm invokes the $instantiateTask$. This task randomly chooses a method m from all methods that are provided by C or by classes in $\mathcal{A}$ and that have C or a subtype of C as return type. If calling m requires arguments, the task invokes the $argumentTask$ to make these arguments available. The $instantiateTask$ returns a call sequence that creates all required arguments, stores them into output variables, and calls m. If the test generator cannot instantiate the CUT, for example, because there is no public constructor, the algorithm fails.

After instantiating the CUT, the algorithm tries to invoke methods on the OUT to change state of the OUT. The $callTask$ randomly chooses a method among all methods of C, invokes the $argumentTask$ for each required argument, and calls the chosen method.

Creating Suffixes During the second step (lines 21 to 26), the algorithm creates a new suffix s_1 for the prefix p by repeatedly invoking the $callTask$. In addition to the suffix, the call in line 23 passes the prefix to the task so that $callTask$ can use output variables from the prefix as input variables in the suffix. After appending calls to the OUT, the new suffix s_1 is added to the set of suffixes for the prefix p.

Tasks that append method calls depend on the $argumentTask$ for getting method arguments. This task uses three strategies to make an argument of a particular type t available. If the given call sequence already has one or more output variables of type t or a subtype of t, the task randomly chooses between reusing a variable and creating a fresh variable. The rationale for creating fresh variables is to diversify the generated tests instead of passing the same arguments again and again. To reuse a variable, the task randomly chooses from all available variables with matching type. To create a fresh variable, the behavior depends on the type t. If t is a primitive type or `String`, the task returns a randomly created literal. Otherwise, the tasks randomly chooses a method from all methods that are provided by C or a class in $\mathcal{A}$ and that return t or a subtype of t. If calling this method requires arguments, the task recursively invokes itself. We limit the number of recursions to avoid infinite loops and fail the task if the limit is reached. If there is no method providing t, the $argumentTask$ returns `null`. Using `null` as an argument may be illegal. In this case, executing the sequence may result in an exception, for example, a `NullPointerException`, so that the candidate is rejected.

Creating Tests The third step of the algorithm (lines 27 to 29) creates tests by combining the new suffix with each existing suffix for the prefix. The algorithm stores the created tests in $\mathcal{T}$ and on future invocations returns a randomly selected test from $\mathcal{T}$ until $\mathcal{T}$ becomes empty.

2.4 Thread Safety Oracle

This section presents an automatic technique to determine whether the execution of a concurrent test exposes a thread safety violation.

2.4.1 Thread Safety

A class is said to be thread-safe if multiple threads can use it without synchronization *and* if the behavior observed by each thread is equivalent to a linearization of all calls [61, 4]. Saying that a class is thread-safe means that all its methods are thread-safe. Alternatively, our approach can also deal with classes that guarantee thread safety for a subset of all methods by excluding the unsafe methods when generating suffixes for a test.

Figure 2.3 is an example for a thread-unsafe class and a test execution that exposes this property. Concurrently using `java.util.ArrayList` results in an exception that does not occur in any of the three possible linearizations of the calls:

`add("a")` → `add("b")` → `hashCode()`	succeeds
`add("a")` → `hashCode()` → `add("b")`	succeeds
`hashCode()` → `add("a")` → `add("b")`	succeeds

That is, the execution in Figure 2.3 shows that `ArrayList` is thread-unsafe, which is not surprising because the class is not supposed to be thread-safe.

A property related to thread safety is atomicity, that is, the guarantee that a sequence of operations performed by a thread appears to execute without interleaved

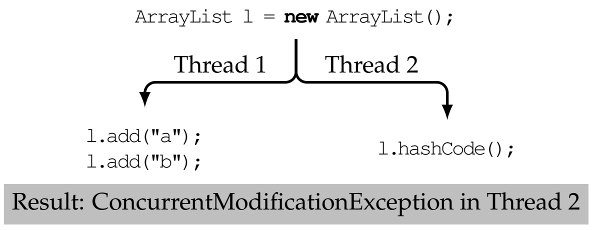

ArrayList l = new ArrayList();
Thread 1
Thread 2
l.add("a");
l.add("b");
l.hashCode();
Result: ConcurrentModificationException in Thread 2

Figure 2.3: Test execution using a thread-unsafe class.

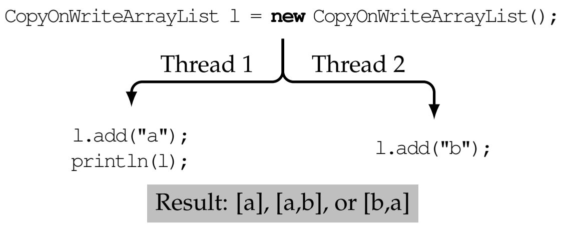

CopyOnWriteArrayList l = new CopyOnWriteArrayList();
Thread 1
Thread 2
l.add("a");
println(l);
l.add("b");
Result: [a], [a,b], or [b,a]

Figure 2.4: Test execution using a thread-safe class.

operations by other threads. One way to make a class thread-safe is to guarantee that each call to one of its methods appears to be atomic for the calling thread. However, a thread-safe class does *not* guarantee that multiple calls to a shared instance of the class are executed atomically [142]. For example, consider the usage of the thread-safe class `java.util.concurrent.CopyOnWriteArrayList` in Figure 2.4. Executing the concurrent test has three possible outputs. For all three, there is a linearization of calls that provides the same result, so the test execution does not expose a thread safety violation:

`add("a")` $\rightarrow$ `println(l)` $\rightarrow$ `add("b")`	gives [a]
`add("a")` $\rightarrow$ `add("b")` $\rightarrow$ `println(l)`	gives [a,b]
`add("b")` $\rightarrow$ `add("a")` $\rightarrow$ `println(l)`	gives [b,a]

2.4.2 Definitions

The thread safety oracle answers the question whether a concurrent test execution exposes a thread safety violation by comparing the concurrent execution to executions of linearizations of the test. We use $\oplus$ to concatenate sequences, and $c \rightarrow_s c'$ indicates that call c comes before call c' in a call sequence s.

Definition 1 (Linearization). *For a test (p, s_1, s_2), let $\mathcal{P}_{12}$ be the set of all permutations of the call sequence $s_1 \oplus s_2$. The set of linearizations of the test is:*

$$\begin{aligned}\mathcal{L}_{(p,s_1,s_2)} = \{p \oplus s_{12} \mid s_{12} \in \mathcal{P}_{12} \wedge \\ (\forall c, c' \;\; (c \rightarrow_{s_1} c' \Rightarrow c \rightarrow_{s_{12}} c') \wedge \\ (c \rightarrow_{s_2} c' \Rightarrow c \rightarrow_{s_{12}} c'))\}\end{aligned}$$

That is, a linearization of a test (p, s_1, s_2) appends to p all calls from s_1 and from s_2 in a way that preserves the order of calls in s_1 and s_2.

Definition 2 (Execution). *For a test (p, s_1, s_2), we denote the set of all executions of this test as $\mathcal{E}_{(p,s_1,s_2)}$. Each $e_{(p,s_1,s_2)} \in \mathcal{E}_{(p,s_1,s_2)}$ represents the sequential execution of p followed by a concurrent execution of s_1 and s_2. Likewise, we denote the sequential execution of a call sequence s as e_s.*

A single test can have multiple executions because concurrent executions are non-deterministic.

The following definition of thread safety refers to the equivalence $e_1 \cong e_2$ of two executions e_1 and e_2. We discuss in Section 2.4.3 how our oracle decides whether two executions are equivalent.

Definition 3 (Thread safety). *Let $\mathcal{T}_C$ be the set of all possible tests for a class C. C is thread-safe if and only if:*

$$\forall (p, s_1, s_2) \in \mathcal{T}_C \;\; \forall e_{(p,s_1,s_2)} \in \mathcal{E}_{(p,s_1,s_2)} \exists l \in \mathcal{L}_{(p,s_1,s_2)} \text{ so that } e_{(p,s_1,s_2)} \cong e_l$$

That is, a class is thread-safe if each concurrent test execution has an equivalent linearization.

2.4.3 The Test Oracle

Showing that a class is thread-safe according to Definition 3 is difficult in practice because all possible tests and all possible executions of these tests would have to be considered. However, the thread safety oracle can show that a class C is thread-unsafe by showing:

$$\exists (p, s_1, s_2) \in \mathcal{T}_C \;\; \exists e_{(p,s_1,s_2)} \in \mathcal{E}_{(p,s_1,s_2)} \text{ so that } \forall l \in \mathcal{L}_{(p,s_1,s_2)} \;\; e_{(p,s_1,s_2)} \not\cong e_l$$

That is, the thread safety oracle tries to find a test that exposes behavior not possible with any linearization of the test. To decide whether two executions are equivalent, the oracle compares the exceptions and deadlocks caused by the executions.

Definition 4 (Equivalence of executions). *Two executions e_1 and e_2 are equivalent if*

- *neither e_1 nor e_2 results in an exception or a deadlock, or*
- *both e_1 and e_2 fail for the same reason (that is, the same type of exception is thrown or both executions end with a deadlock).*

Although this abstraction ignores many potential differences between executions, such as different return values of method calls, Definition 4 is crucial to ensure that the analysis only reports a class as thread-unsafe if the class is definitely thread-unsafe.

Algorithm 2 Checks whether a test (p, s_1, s_2) exposes a thread safety bug

```
1: repeat
2:     $e_{(p,s_1,s_2)} \leftarrow execute(p, s_1, s_2)$
3:     if $failed(e_{(p,s_1,s_2)})$ then
4:         $seqFailed \leftarrow$ false
5:         for all $l \in \mathcal{L}(p, s_1, s_2)$ do
6:             if $seqFailed =$ false then
7:                 $e_l \leftarrow execute(l)$
8:                 if $failed(e_l) \wedge sameFailure(e_{(p,s_1,s_2)}, e_l)$ then
9:                     $seqFailed \leftarrow$ true
10:        if $seqFailed =$ false then
11:            report violation $e_{(p,s_1,s_2)}$ and exit
12: until $maxConcExecs$ reached
```

Algorithm 2 shows how the analysis checks whether a test (p, s_1, s_2) exposes a thread safety problem. The algorithm repeatedly executes the test until a maximum number of concurrent executions is reached, or until a thread safety violation is found. If the test fails, that is, it throws an exception or results in a deadlock, the algorithm executes all linearizations of the test to check whether the same failure occurs during a sequential execution of the same calls. If no linearization exposes the same failure, the algorithm reports a thread safety violation.

The thread safety oracle is sound and incomplete.[5] Every execution for which the oracle reports a bug is guaranteed to expose a thread safety violation according to Definition 3, but the oracle may classify an execution as correct even though it exposes a thread safety problem. The soundness of the oracle ensures that our approach detects concurrency bugs without reporting false positives.

We build upon two assumptions, which we find true for most real-world classes during our evaluation. First, we assume that uncaught exceptions and deadlocks that occur in a concurrent usage of a class but not in a sequential usage are considered a problem. This assumption is in line with the commonly accepted definition of thread safety [61, 4]. Second, we assume that sequential executions of a call sequence behave deterministically. Sequentially non-deterministic methods, for example, methods that depend on the current system time, should be excluded from our analysis. Alternatively, our analysis can be combined with a runtime environment that ensures deterministic sequential execution [122]. For concurrent executions, the analysis supports non-deterministic behavior by repeatedly executing concurrent tests to trigger different interleavings.

2.5 Implementation

We implement the test generator and the thread safety oracle into an automatic and precise bug detection tool for thread-safe Java classes. This section presents several challenges faced by the implementation and how we address them.

[5]We mean soundness and completeness regarding incorrectness [144]. In other contexts, such as type systems, the terms are typically used with respect to correctness, that is, inverse to the usage here.

The test generator executes many call sequences and must ensure that each call sequence executes efficiently and without influencing later executions of call sequences. To execute sequences efficiently we take a reflection-based approach similar to the sequential test generator Randoop [117]. A problem not addressed by Randoop is that sequentially executed call sequences may influence each other because of static state. This problem is independent of concurrency. For example, a call sequence s_1 may assign a value to a static field and a call sequence s_2 may read the static field. As a result, the outcome of executing s_2 may vary depending on whether s_1 is executed before s_2. We address the problem by resetting all static state of a class before each execution of a call sequence. For this purpose, our implementation instruments classes so that all modifications of static state are recorded and can be reset to the state just after loading the class. Csallner and Smaragdakis describe a similar approach for a sequential test generator [36].

The test generator takes a random seed as an input, which allows for replaying the test generation process by using the same seed again, as long as the tested classes behave deterministically in sequential executions. Experience with sequential, random test generation shows that many short runs with different seeds trigger bugs faster than few long runs [31]. Our initial experiments confirmed this observation, so we run the analysis in multiple rounds that each use a different random seed. The first rounds stop after trying a small number of suffixes (ten) for a single prefix. Later rounds gradually raise the maximum number of generated prefixes ($maxPrefixes$) to 25 and the maximum number of generated suffixes to 500. The values of other parameters used in Algorithms 1 and 2 are $maxStateChangerTries = 5$, $maxConcExecs = 100$, and $maxCallTries = 2$.

To detect deadlocks, we use the management interface of the thread system of the Java virtual machine. A daemon thread periodically queries this interface and notifies the thread safety oracle in case of a deadlock.

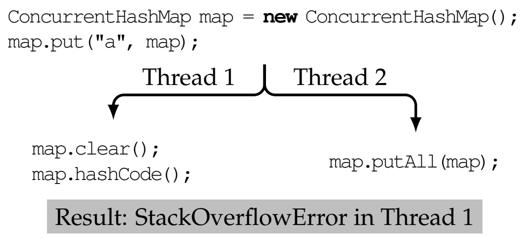
2.6 Evaluation

We evaluate our approach by applying the prototype implementation to classes from six popular code bases: the Java standard library shipped with Sun's JDK, the database connection pool library Apache Commons DBCP, the serialization library XStream, the text processing toolkit LingPipe, the chart library JFreeChart, and Joda-Time, a library to handle data and time. We select these classes based on documentation claiming that a class is thread-safe and based on known bug reports related to thread safety.

2.6.1 Experimental Setup

All experiments are done on an eight-core machine with 3GHz Intel Xeon processors and 8GB memory running 32-bit Ubuntu Linux and the Java Hotspot VM version 1.6.0_27, giving 2GB memory to the VM. We run experiments with different CUTs in parallel but run at most four tests at a time to reserve a core for each concurrent thread exercising the CUT. We repeat each experiment with ten different random seeds [117].

To analyze a CUT, we give auxiliary classes to the test generator, namely all other public classes from the code base and common classes from the Java standard library.



```
ConcurrentHashMap map = new ConcurrentHashMap();
map.put("a", map);
```

Thread 1:

```
map.clear();
map.hashCode();
```

Thread 2:

```
map.putAll(map);
```

Result: StackOverflowError in Thread 1

(a) Execution of a generated concurrent test that exposes a thread safety violation.

```
class ConcurrentHashMap {
  void clear() {
    final Segment[] segments = this.segments;
    for (int j = 0; j < segments.length; ++j) {
      Segment s = segmentAt(segments, j);
      if (s != null) s.clear(); // locks the segment
    }
  }
  void putAll(Map m) {
    for (Map.Entry e : m.entrySet())
      // BUG: m's entries may change
      put(e.getKey(), e.getValue());
  }
  Object put(Object key, Object value) {
    Segment s = /* get segment in a thread-safe way */;
    return s.put(key, value); // locks the segment
  }
}
```

(b) Supposedly thread-safe class.

Figure 2.5: Concurrency bug in `ConcurrentHashMap`.

2.6.2 Bugs Found

The analysis finds 15 bugs in supposedly thread-safe classes, six of them previously unknown. Each bug can cause a crash in a concurrent program that relies on the thread-safe class. Table 2.1 lists all bugs along with the reason for failing. The last column indicates whether the reason is an exception thrown explicitly in the analyzed code base, or whether it is an exception thrown by the Java standard library or a failure triggered implicitly by the virtual machine. For twelve of 15 bugs, a failure triggered by the standard library or by the virtual machine is sufficient to reveal the bug. That is, our approach reveals most bugs without requiring the analyzed classes to throw an exception if an unsafe state is reached.

The analysis reveals two previously unknown bugs in the Java standard library, one of them shown in the detailed example in Section 2.2. The other previously un-

Table 2.1: Summary of results. The last column indicates whether the reason for failing is an exception explicitly thrown in the code base under test, or if it is an exception thrown by the Java standard library or an implicit failure triggered by the virtual machine.

ID	Code base	Class	Declared thread-safe	Found unsafe	Reason for failing	Thrown by CUT
Previously unknown bugs:						
(1)	JDK 1.6.0_27 and 1.7.0	StringBuffer	yes	yes	IndexOutOfBoundsException	no
(2)	JDK 1.6.0_27 and 1.7.0	ConcurrentHashMap	yes	yes	StackOverflowError	no
(3)	Commons DBCP 1.4	SharedPoolDataSource	yes	yes	ConcurrentModificationException	no
(4)	Commons DBCP 1.4	PerUserPoolDataSource	yes	yes	ConcurrentModificationException	no
(5)	XStream 1.4.1	XStream	yes	yes	NullPointerException	no
(6)	LingPipe 4.1.0	MedlineSentenceModel	yes	yes	IllegalStateException	yes
Previously known bugs:						
(7)	JDK 1.1	BufferedInputStream	yes	yes	NullPointerException	no
(8)	JDK 1.4.1	Logger	yes	yes	NullPointerException	no
(9)	JDK 1.4.2	SynchronizedMap	yes	yes	Deadlock	no
(10)	JFreeChart 0.9.8	TimeSeries	yes	yes	NullPointerException	no
(11)	JFreeChart 0.9.8	XYSeries	yes	yes	ConcurrentModificationException	no
(12)	JFreeChart 0.9.12	NumberAxis	yes	yes	IllegalArgumentException	yes
(13)	JFreeChart 1.0.1	PeriodAxis	yes	yes	IllegalArgumentException	yes
(14)	JFreeChart 1.0.9	XYPlot	yes	yes	ConcurrentModificationException	no
(15)	JFreeChart 1.0.13	Day	yes	yes	NumberFormatException	no
Automatic classification of classes as thread-unsafe:						
(16)	Joda-Time 2.0	DateTimeFormatterBuilder	no	yes	IndexOutOfBoundsException (10x)	no
(17)	Joda-Time 2.0	DateTimeParserBucket	no	yes	IllegalArgumentException (9x)	yes
					NullPointerException (1x)	no
(18)	Joda-Time 2.0	DateTimeZoneBuilder	no	yes	NullPointerException (6x)	no
					ArrayIndexOutOfBoundsException (2x)	no
					IllegalFieldValueException (2x)	no
(19)	Joda-Time 2.0	MutableDateTime	no	yes	IllegalFieldValueException (9x)	yes
					ArithmeticException (1x)	no
(20)	Joda-Time 2.0	MutableInterval	no	yes	IllegalArgumentException (10x)	yes
(21)	Joda-Time 2.0	MutablePeriod	no	yes	ArithmeticException (10x)	no
(22)	Joda-Time 2.0	PeriodFormatterBuilder	no	yes	ConcurrentModificationException (5x)	no
					IndexOutOfBoundsException (4x)	no
					IllegalStateException (1x)	yes
	Joda-Time 2.0	ZoneInfoCompiler	no	no	(stopped after 24h)	–

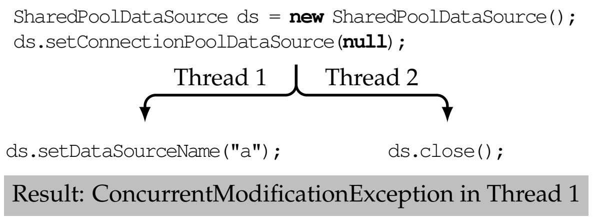
known bug in the Java standard library is illustrated in Figure 2.5. The class `ConcurrentHashMap` is part of the `java.util.concurrent` package, which provides thread-safe collection classes. For better scalability, the class divides the map into segments that are locked independently of each other, instead of relying on a single exclusion lock. Unfortunately, `putAll()` does not consider the case where the passed map is the same as the receiver object of the call. The method retrieves the entries of the passed map without any synchronization and then passes each element to the correctly synchronized `put()`. As a result, a call to `map.putAll(map)` can undo changes done by a concurrently executing thread that also modifies the map—a clear thread safety violation. Our analysis generates the test in Figure 2.5a, which exposes the problem. Calling `hashCode()` results in a stack overflow for self-referential collections because the method recursively calls itself (this is documented behavior). If `ConcurrentHashMap` were thread-safe, this behavior should not be triggered by the test because all three possible linearizations of the calls in Figure 2.5a call `clear()` before calling `hashCode()`. However, the test fails with a `StackOverflowError` because `putAll()` undoes the effects of the concurrently executed `clear()`.

Figure 2.6 shows a previously unknown bug that our analysis detects in Apache Commons DBCP. The supposedly thread-safe class `SharedPoolDataSource` provides two methods `setDataSourceName()` and `close()` that register and deregister the data source via static methods of `InstanceKeyObjectFactory`, respectively. The factory class maintains a thread-unsafe `HashMap` that assigns names to data sources. Although registering new instances is synchronized to avoid concurrent accesses to the `HashMap`, deregistering instances is not synchronized. The generated test in Figure 2.6a shows that this lack of synchronization leads to an exception when calling `setDataSourceName()` and `close()` concurrently.

2.6.3 Annotating Classes as Thread-unsafe

Beyond finding bugs in thread-safe classes, our analysis can be used to analyze classes that do not have any documentation on their thread safety and to automatically annotate these classes as thread-unsafe. In a preliminary study for this work, we found that one of the most common concurrency-related questions of Java developers is whether a particular library class is thread-safe. Few libraries come with precise documentation to answer this question. To address this lack of documentation, our analysis can automatically annotate classes as thread-unsafe. Since the thread safety oracle is sound (Section 2.4), these annotations are guaranteed to be correct.

To evaluate this usage scenario, we run our analysis on a library that specifies for each class whether it is thread-safe or not. The library (Joda-Time) contains 41 classes, of which 33 are documented as thread-safe and eight are documented as thread-unsafe. The lower part of Table 2.1 summarizes the results. For seven of the eight thread-unsafe classes, our analysis detects a thread safety problem. The missing class, `ZoneInfoCompiler`, reads files from the file system, transforms them, and writes other files as output. Since the thread safety oracle does not check the integrity of such files, it cannot detect problems caused by concurrent usages of the class. For the 33 classes that are documented as thread-safe, no problems have been found after running the analysis for 24 hours.



```
SharedPoolDataSource ds = new SharedPoolDataSource();
ds.setConnectionPoolDataSource(null);
```

```
ds.setDataSourceName("a");          ds.close();
```

Result: ConcurrentModificationException in Thread 1

(a) Execution of a generated concurrent test that exposes a thread safety violation.

```
class SharedPoolDataSource {
  void setDataSourceName(String v) {
    key = InstanceKeyObjectFactory.registerNewInstance(this);
  }
  void close() {
    InstanceKeyObjectFactory.removeInstance(key);
  }
}

class InstanceKeyObjectFactory {
  static final Map instanceMap = new HashMap();
  synchronized static String registerNewInstance(SharedPoolDataSource ds) {
    // iterate over instanceMap
  }
  static void removeInstance(String key) {
    // BUG: unsynchronized access to instanceMap
    instanceMap.remove(key);
  }
}
```

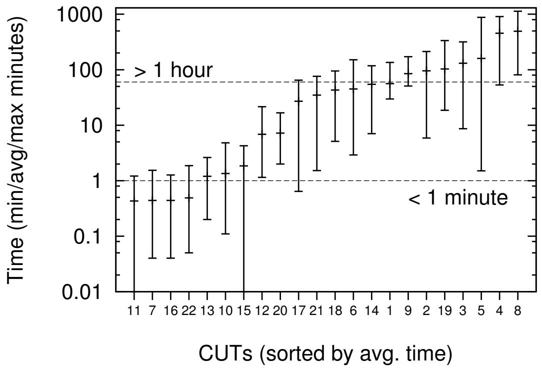
(b) Supposedly thread-safe class.

Figure 2.6: Concurrency bug in Apache Commons DBCP.

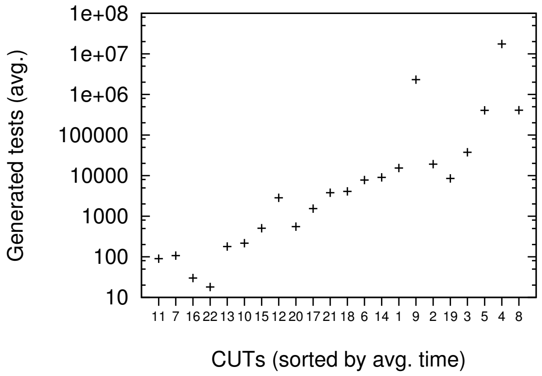
2.6.4 Performance

Figure 2.7a shows how long the analysis takes to trigger the problems from Table 2.1. The horizontal axis shows the IDs of CUTs from Table 2.1. The vertical axis gives the minimum, average, and maximum time taken to find the violation over ten runs. Most of the problems are found within one hour. For several classes, the analysis takes only a few seconds. Other classes require several hours of computation time, with up to 8.2 hours for bug 8 (JDK's `Logger`). Given that the bug remained unnoticed for several years in one of the most popular Java libraries, we consider this time to be still acceptable. Section 2.8 outlines approaches to reduce the running time of our implementation with additional engineering effort.

A question related to running time is how many tests the analysis generates and executes before hitting a bug. Figure 2.7b shows the average number of generated



(a) Time to detect thread safety violations.



(b) Tests generated before triggering a thread safety problem.

Figure 2.7: Computational effort required to detect thread safety violations.

tests for each bug, listing the bugs in the same order as in Figure 2.7a. For some bugs, a small number of tests (several hundreds or even less than hundred) suffices to expose the problem. Other bugs require more tests, up to 17 million for bug 4. A manual inspection of bugs requiring many tests suggests that the task of executing a bug-exposing test with the "right" thread interleaving dominates over the task of generating a bug-exposing test.

The approach has very moderate memory requirements. The test generator selects methods randomly and therefore does not maintain significant amounts of state. If executing a generated call sequence exceeds the available memory, an exception is thrown and the sequence is not extended.

2.6.5 Threats to Validity

There are a number of potential threats to the validity of the evaluation. First, the selection of classes we analyze may not be representative for thread-safe Java classes and another set of classes may give different results. To address this threat, we select classes from well-tested code bases. We suspect that classes in an earlier stage of development contain at least as many thread safety problems as the classes we analyze. Second, the parameters of the test generator, such as *maxStateChangerTries*, may influence the results. We set these parameters based on our intuition and the results from initial experiments, and therefore, we cannot guarantee them to be optimal in any sense. Third, the set of auxiliary classes provided to the test generator influences the ability of the analysis to find thread safety violations, and we may accidentally have selected auxiliary classes that bias our results. Fourth, the analysis has two major sources of non-determinism: the random-based test generator and the non-deterministic Java scheduler. The number of repetitions we use for measuring performance may not be sufficient to accurately characterize these random processes. Also, performing the experiments on a different platform than the one we use may change the results because scheduling decisions may differ. Finally, the assessment that all reported warnings are true positives depends on the assumption that the analyzed classes are supposed to be thread-safe. If this safety property is not desired, then the warnings may not be relevant for developers. To mitigate this threat, we select classes that are explicitly documented as thread-safe.

2.7 Support for the Thesis

The analysis presented in this chapter supports our thesis that automatic program analysis allows for precisely detecting programming errors with little effort.

2.7.1 Automation

The analysis is fully automatic because all input that the analyses requires is the supposedly thread-safe class under test, possibly accompanied by auxiliary classes that the test generator may need, for example, to create method arguments. In practice, we find other classes from the same library or program to be sufficient as auxiliary classes, so that the analysis can be run fully automatically.

2.7.2 Precision

Each warning reported by the analysis corresponds to a thread safety violation, which—given that the CUT is supposedly thread-safe—is certainly a bug. The thread safety oracle ensures this property by focusing on certainly undesired behavior, exceptions and deadlocks, and by comparing concurrent tests to their linearizations.

2.7.3 Effort

Using the analysis involves minimal human and reasonable computational effort. Human effort is minimal because the input to the analysis is available for every piece of compilable software (simply because it is the software itself), and because all output of the analysis is relevant for developers. The performance of our prototype implementation shows that the analysis finds bugs in an acceptable amount of time (Section 2.6.4), in particular, given that the analysis runs without human interaction.

2.8 Limitations and Future Work

Directions of future work include:

- The current approach has limited support for multi-object bugs. Although the generated tests combine multiple classes and objects, these tests only reveal bugs triggered by concurrent calls to a single object. Exploring ways to extend the current analysis to groups of interacting objects is a promising line of future work.
- The approach is limited to bugs that manifest through an exception or deadlock, and therefore the approach may miss more subtle problems that lead to incorrect but not obviously wrong results. The challenge is to address this limitation while maintaining the precision of the analysis.
- The current approach assumes that exceptions and deadlocks are undesirable. We found this "implicit specification" to be true in practice, but in principle a class could throw exceptions as part of its legal, concurrent behavior. Future work may identify exceptions that are part of the expected concurrent behavior, for example, by mining the documentation of the CUT or by analyzing executions of programs that use the CUT.
- The test generator may not be able to generate reasonable input for classes that rely on a particular environment, such as a database connection, or that require complex input, such as valid HTML code. One way to address this limitation could be to extract code sequences from existing programs that use a class and to incorporate these code sequences into generated tests.
- Future work may improve the performance of our implementation. The by far most important bottleneck in the current implementation are the repeated concurrent executions of tests. For example, for the CUT that requires most time during our evaluation, 99.5% of the time is spent with concurrent executions. We envision two approaches to address this problem. First, the analysis can exploit multiple cores by exploring different concurrent executions of the same test in parallel. Second, our analysis can be easily combined with existing techniques to increase the probability of hitting a bug by controlling or perturbing the scheduler [42, 140, 110, 35, 19]. Our current implementation executes tests with the standard Java thread scheduler. To plug a more sophisticated scheduling technique into our approach, one can redefine the $execute()$ function of Algorithm 2.
- Generating concurrent tests allows automating a wide range of existing dynamic analyses. For example, executions of generated tests can be analyzed by existing detectors of data races [139, 50, 104, 141] or atomicity violations [51, 67, 120, 86, 142]. Without generated tests, these analyses are limited to manually created tests and the bugs exposed by these tests.

Unsafe Substitutes 3

With great power comes great responsibility. When creating a subclass, programmers have great power: They can add, remove, and modify behavior. Even though statically typed languages, such as Java, restrict this power on the type level, programmers can create subclasses that modify superclass behavior on the semantic level. For example, a subclass can replace a method that performs a complex computation with an empty method, replace a pure method with a non-pure one, or replace a method that succeeds for some input with a method that fails for this input.

The responsibility that comes with this power is to ensure *substitutability* [94]: A subclass instance should behave like a superclass instance when being referred to through the superclass type. The reason for this requirement is that a polymorphic reference of a superclass type can point to a subclass instance and that the programmer may not know the runtime type of the reference. Substitutability is commonly accepted as one of the bedrock principles of object-oriented programming. We call a subclass that fulfills the substitutability requirement a *safe substitute* for its superclass and an *unsafe substitute* otherwise.

Although substitutability is crucial for object-oriented programming, developers get little help from existing languages and tools to ensure that subclasses are safe substitutes. Popular languages, such as Java, enforce substitutability only on the type level but not on a deeper, semantic level. While the type system ensures that an overriding method is type-compatible with the overridden method, it does not guarantee that the overriding method semantically substitutes the overridden method. Specification and verification techniques to ensure substitutability through behavioral subtyping have been proposed [90, 8, 95, 41, 137], but none of them has found its way into mainstream programming. Furthermore, we are not aware of any automatic and precise tool that addresses the problem of ensuring substitutability.

This chapter presents an automatic and precise bug detection approach that finds unsafe substitutes. The approach is automatic because to test whether a subclass is a safe substitute for a superclass it suffices to provide these two classes, possibly together with other classes on which the superclass and subclass may depend. The approach is precise because all reported warnings point to classes that are unsafe substitutes.[1]

[1]Precision is guaranteed only to the degree that constructor mappings (Section 3.3.1) are precise. During our evaluation, all unsafe substitutes that are reported because they cause a crash are true positives.

The following section motivates the problem of revealing unsafe substitutes. Section 3.2 outlines our automatic and precise bug detection approach. The details of the approach are presented in Sections 3.3 to 3.5. In Section 3.6, we present the results from applying the approach to real-world Java classes. Section 3.7 discusses how this chapter supports the thesis of this work, and Section 3.8 outlines directions of future work.

3.1 Motivation

The need to ensure substitutability is the result of combining two language features that are part of most class-based, object-oriented programming languages: polymorphism and subtyping via subclassing. Polymorphism allows a reference of type $Super$ to point to an object of any subtype Sub of $Super$. Polymorphic references that point to subtype instances occur in various situations, for example, when a method with declared return type $Super$ returns a Sub instance, or when a Sub argument is passed to a method expecting a $Super$ parameter, or when a Sub instance is stored in a field of type $Super$. Subclassing and subtyping are merged into a single concept in most popular languages. That is, a programmer that creates a subclass not only creates a new class but also creates a new subtype of the superclass type.

The ubiquity of polymorphic references combined with subclasses that create subtypes leads to a problem: A subclass Sub that behaves differently from its superclass $Super$ can surprise a client of $Super$ that unconsciously works with an instance of Sub. For example, consider a client that calls a method on a variable with static type $Super$, where the variable is the return value of a call to a third-party library. The client cannot foresee the runtime type of the returned object and decides to call the method based on the variable's static type. If the behavior of the call depends on the object's runtime type, the programmer is caught off guard and may have to deal with a bug that she is not responsible for and that she cannot foresee.

To avoid surprises due to subtype instances that are hidden by polymorphic references, a subclass should not diverge from the superclass behavior. This idea, coined as substitutability, has become textbook knowledge [148, 57] and is widely accepted as one of the core principles of object-oriented programming. Substitutability is crucial for object-oriented programming because it allows a programmer to reason about the behavior of an object based on the object's statically declared type, that is, without knowing its runtime type. What if a programmer that uses objects of declared type *Super* could not assume subclasses to be safe substitutes? A prudent programmer would have to study all subclasses of *Super*, figure out which of them are compatible with the intended usage, and add runtime type checks before each use of an object with an unknown runtime type. Since this approach would add boilerplate code and runtime overhead, it obviously reduces the benefits of subclassing and polymorphism. Instead, substitutability allows programmers to use a polymorphic reference without knowing its runtime type.

Despite the importance of substitutability, programmers do not have any tools to test whether subclasses are safe substitutes. As a result of this lack of testing tools, programmers are susceptible to creating subclasses that are unsafe substitutes. For example, consider `FastTreeMap` from the Apache Commons Collections library. `FastTreeMap` is a subclass of `TreeMap` and therefore should behave like its superclass when being referred to as a `TreeMap`. Figure 3.1 shows a usage of these classes

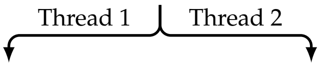
```
TreeMap m = new TreeMap() OR new FastTreeMap();
m.put(23, m);
m.pollLastEntry();
m.hashCode();
```

Result:

```
                 OK  if m is TreeMap
StackOverflowError  if m is FastTreeMap
```

Figure 3.1: Sequential usage that shows `FastTreeMap` to be an unsafe substitute.

```
Properties p = new Properties() OR new PropertyMap();
p.setProperty("a", "b");
```



```
p.remove("a");                 p.clear();
```

Result:

```
                   OK  if p is Properties
NullPointerException  if p is PropertyMap
```

Figure 3.2: Concurrent usage that shows `PropertyMap` to be an unsafe substitute.

that reveals that `FastTreeMap` is an unsafe substitute for `TreeMap`.[2] Executing the code succeeds if `m` is a `TreeMap` but raises an exception if `m` is a `FastTreeMap`. This difference may crash a client program that expects a `TreeMap` reference to behave like a `TreeMap`. We reported this problem to the developers, who acknowledged our report.[3]

Substitutability, which traditionally has been considered in sequential programs, is equally important in concurrent programs. Similar to the sequential setting, a subclass instance referenced through a superclass type and shared by multiple concurrent threads should behave like a superclass instance. In particular, subclasses of a thread-safe class must be thread-safe as well. That is, if multiple threads are allowed to call methods of a shared object without synchronization, then this property must be preserved by subclasses.

Unfortunately, existing languages and tools do not support programmers in ensuring substitutability for concurrently used classes. Figure 3.2 shows a substitutability problem in `PropertyMap` from JBoss. The class extends `Properties`, a thread-safe class from the Java standard library, and therefore, should also support concurrent usage. However, `PropertyMap` causes an exception in a concurrent usage that

[2]We use the syntax `new A() OR new B()` to say that either of the constructors can be executed.

[3]See issue 394 in the Commons Collections bug tracker. The class will be removed in the next version of the library.

succeeds with the superclass. We reported this problem to the developers, who fixed the bug.[4]

3.2 Overview of the Approach

How can developers find unsafe substitutes, such as Figures 3.1 and 3.2? In the following, we present an automatic and precise analysis to reveal subclasses that are unsafe substitutes. The input to our analysis are a subclass Sub and a superclass $Super$, possibly accompanied by classes that the other classes depend on. The output of the analysis are warnings that show with concrete test cases how using a subclass instance through the superclass type leads to visible behavior that is impossible with a superclass instance. The analysis checks substitutability for sequentially and concurrently used classes and, among many others, finds the bugs in Figures 3.1 and 3.2.

Our approach to reveal unsafe substitutes combines two techniques (Figure 3.3). First, a *generator of generic tests* creates tests that exercise sequential and concurrent behavior (Section 3.3). The generated tests are generic in the sense that they can test the behavior of Sub as well as the behavior of $Super$. Generic tests simulate how clients of Sub and $Super$ use these classes. The test generator considers only public methods, fields, and constructors because these are accessible to clients.

Second, an analysis called the *superclass oracle* checks for each test whether the test exposes behavior that occurs when using Sub but that cannot occur when using $Super$ (Section 3.4). If the superclass oracle detects such behavior, it reports a warning to the developer. By construction, each warning comes with a concrete test case showing how the subclass behavior diverges from the superclass behavior.

Before explaining the details of the approach, we define when a subclass is a safe substitute for a superclass and describe two ways in which a subclass can be an unsafe substitute. When saying subclass, we mean a direct or indirect subclass relationship. The definitions build upon the notion of a *usage*, which describes how a class

[4]See issue 126 in the JBoss Common bug repository.

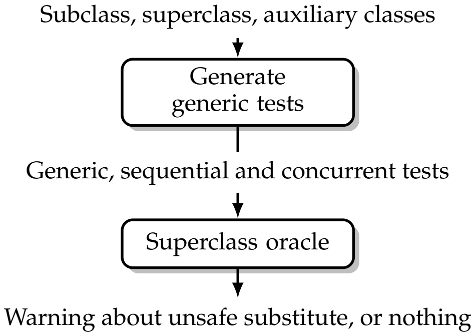

Subclass, superclass, auxiliary classes
Generate generic tests
Generic, sequential and concurrent tests
Superclass oracle
Warning about unsafe substitute, or nothing

Figure 3.3: Overview of the approach.

is used in a client program. We refer to usages that apply to both a superclass and a subclass as *generic usages*.

Definition 5 (Generic usage)**.** *A generic usage with respect to classes $Super$ and Sub is a partial order of constructor calls, method calls, and field accesses that use an object of runtime type $Super$ or Sub through the interface offered by $Super$. The object is called the object under test (OUT).*

For a generic usage u, we write u_{Super} if the OUT has runtime type $Super$ and u_{Sub} if the OUT has runtime type Sub.

A generic usage can describe both sequential and concurrent usages of objects. For a sequential usage, the partial order of calls and field accesses is a total order. For a concurrent usage, the partial order reflects multiple concurrent threads.

Definition 6 (Safe substitute)**.** *A subclass Sub is a safe substitute for a superclass $Super$ if and only if for each generic usage u, the visible behavior of u_{Super} is equivalent to the visible behavior of u_{Sub}.*

By visible behavior we mean application-level behavior, such as return values of method calls, text written to the console, network packets sent to other machines, and exceptions thrown. We ignore machine-level behavior, such as memory accesses and CPU usage. In the following, we define two kinds of unsafe substitutes, which differ in the way the visible behavior of the subclass diverges from the visible behavior of the superclass.

Definition 7 (Output-diverging (unsafe) substitute)**.** *A subclass Sub is an output-diverging substitute for a superclass $Super$ if and only if there exists a generic usage u so that there exists an output of u_{Sub} that is not an output of u_{Super}.*

By output we mean the sequence of return values of method calls on the OUT. For sequential and deterministic usages, there exists exactly one output per usage. If the subclass output is different from the superclass output, then the subclass is an output-diverging substitute. For concurrent usages, there can be multiple outputs for a single usage, for example, due to scheduling non-determinism. If the set of subclass outputs contains an output that is not in the set of superclass outputs, then the subclass is an output-diverging substitute.

Definition 8 (Crashing (unsafe) substitute)**.** *A subclass Sub is a crashing substitute for a superclass $Super$ if and only if there exists a generic usage u so that u_{Sub} can lead to an uncaught exception or to a deadlock that cannot occur with u_{Super}.*

Of the unsafe substitutes considered here, crashing substitutes are the more severe kind of unsafe substitutes because they lead to certainly undesired behavior in clients using a Sub instance through type $Super$. Output-diverging substitutes may or may not be a problem in practice (discussed in detail in Section 3.6).

3.3 Generating Generic Tests

In the following, we present our approach for generating generic tests for sequentially and concurrently used classes. The test generator builds upon the test generator described in Chapter 2, which creates concurrent tests. To search for unsafe substitutes, we adapt the test generator to generate generic tests and to generate both sequential and concurrent tests. The description in this section is self-contained and sufficiently detailed to understand our analysis for finding unsafe substitutes. Some technical details elaborated on in Chapter 2 are omitted here.

To compare $Super$ and Sub, we require tests that can check the behavior of OUTs of both runtime types Sub and $Super$. We call such a test a *generic test*. In a generic test, the static type of the variable v_{OUT}, which represents the OUT, is $Super$. The dynamic type of v_{OUT} can be either $Super$ or Sub. Using the static type $Super$ for the OUT ensures that each call involving the OUT is type-compatible with both $Super$ and Sub. A generic test has two parts. One part creates the OUT and therefore decides on the OUT's runtime type. This part has two variants, one that instantiates $Super$ and one that instantiates Sub. The other part uses the OUT and is independent of the OUT's runtime type.

3.3.1 Constructor Mappings

Because we compare the behavior of $Super$ and Sub instances, it is crucial that the constructors used for creating these instances are semantically equivalent. Finding pairs of $Super$ and Sub constructors that create semantically equivalent objects is non-trivial in Java and cannot be fully automated for two reasons. First, constructors are not inherited [62]. For example, Sub may have a single constructor expecting an `int` parameter, while $Super$ offers only the parameterless default constructor. Second, calling a Sub constructor that leads to the same inherited state as calling a $Super$ constructor does not guarantee that the resulting objects are semantically equivalent. For example, a superclass may use a field to store a length in meters, whereas the subclass uses the same field to store the length in yards.

To address the problem of finding equivalent constructors, we use *constructor mappings* that specify how calling a constructor of Sub can be transformed into a semantically equivalent call to a constructor of $Super$. We provide a heuristic that automatically generates constructor mappings under the assumption that $Super$ and Sub constructors that expect the same types of arguments are semantically equivalent. Alternatively, a user of our approach can specify constructor mappings explicitly. In our evaluation, we use automatically generated constructor mappings.

For example, consider the following constructor mappings:

$$Sub(Foo, Bar) \rightarrow Super(1, 2)$$

$$Sub(int, boolean) \rightarrow Super(2)$$

The first mapping specifies that calling `new Sub(someFoo, someBar)` is semantically equivalent to calling `new Super(someFoo, someBar)`. In practice, most constructor mappings are similar to the first example, which is why our heuristic generates this mapping automatically. The second mapping specifies that we can replace a call to `new Sub(someInt, someBoolean)` with a call to `new Super(someBoolean)`, that is, we pass the second argument of *Sub*'s constructor as the only argument to *Super*'s constructor and omit the `int` parameter, which is not required by $Super$. If necessary, a user can specify the second mapping in addition to or instead of generated constructor mappings.

3.3.2 Generating Sequential and Concurrent Tests

The test generator can be configured to produce either sequential tests, concurrent tests, or both kinds of tests, depending on the kind of usage the tested class is intended for. A *sequential test* is a sequence of calls executed in a single thread. A *concurrent test* is a partially ordered set of calls that are executed in multiple concurrent threads. We focus on concurrent tests that consist of a *sequential prefix* followed by two *parallel suffixes*, where the prefix is a call sequence executed in a single thread, and where each suffix contains a single method call executed in a separate thread that runs concurrently with the other suffix thread. The rationale for this choice is twofold. First, a recent study of real-world concurrency bugs has shown that most bugs (96%) can be reproduced with two threads [100]. Second, the execution of concurrent suffixes with a single method call can be efficiently explored and still reveals many real-world bugs [113].

To create a generic test, the first step is to generate calls that create the OUT. The generator selects a constructor mapping and generates two call sequences with calls to instantiate $Super$ and Sub, respectively. These two sequences give two variants of the test, which allows us to test both $Super$ and Sub. The second step is to generate the part of the test that uses the OUT. For a sequential test, the generator repeatedly appends calls to the OUT. For a concurrent test, the generator appends calls to the OUT to obtain the prefix of the test, and then creates suffixes by spawning two threads and by adding a call to each of them.

Calling methods often requires method arguments. The test generator randomly chooses arguments, for example, by calling a method that returns an object of the required type. Calls made to obtain arguments for calls in the suffix of a concurrent test are added to the prefix instead of the suffix. For example, to call a method `m(A a)` in the suffix, the test generator adds calls to the prefix to obtain an instance of `A`. The rationale for pushing calls back to the prefix is to keep the suffix as short as possible, which in turn, reduces the time for exploring the interleavings of a concurrent test.

Most decisions of the test generator on how to explore the space of possible tests are taken randomly and based on feedback from executions of partial tests [117]. For example, when choosing which constructor or method to call, or how to obtain an argument, the test generator randomly selects from all available options. Whenever a call is added to a partial test, the test generator executes the extended test and checks whether adding the call leads to an exception or to a deadlock. If adding a call leads to a failure, the call is discarded and another call is added instead. For executing tests during test generation, we use the variant of the test that exercises a Sub instance. The reason is that partial tests that fail with Sub can be passed to the crash oracle (Section 3.4.2), which will check whether the test also fails with $Super$.

Figures 3.1 and 3.2 are examples of generic tests for sequentially and concurrently used classes, respectively. The `new A() OR new B()` syntax expresses the two variants of the tests. Depending on which of the two constructors is used, the tests exercise either a superclass instance or a subclass instance.

3.4 The Superclass Oracle

Given a generic test for two classes $Super$ and Sub, the superclass oracle checks whether the test exposes that Sub is an unsafe substitute for $Super$. The oracle uses $Super$ as an executable specification of the correct behavior of Sub and compares the

Algorithm 3 Output oracle. Checks whether Sub is an output-diverging substitute for $Super$.

```
Input: Generic test $t$ for classes $Super$ and $Sub$
Output: Warning about an unsafe substitute, or nothing
1: $\mathcal{O}_{Super} \leftarrow execute(t_{Super})$
2: $\mathcal{O}_{Sub} \leftarrow execute(t_{Sub})$
3: $\mathcal{O}_{SubOnly} \leftarrow \mathcal{O}_{Sub} - \mathcal{O}_{Super}$
4: if $\mathcal{O}_{SubOnly} \neq \emptyset$ then
5:     $reportWarning(t, \mathcal{O}_{SubOnly})$
```

visible behavior of both classes. The visible behavior of a subclass can diverge in different ways from the visible behavior of a superclass. In the following, we present two variants of the superclass oracle, the output oracle and the crash oracle, which focus on revealing output-diverging substitutes and crashing substitutes, respectively. The input to both oracles is a generic test t. Similar to generic usages, we refer to a test that exercises an OUT of runtime type $Super$ as t_{Super} and likewise for Sub.

3.4.1 The Output Oracle

The output oracle checks whether a generic test exposes that Sub is an output-diverging substitute for $Super$. Algorithm 3 summarizes the approach. The oracle executes both t_{Super} and t_{Sub} and stores the sequences of return values in the output sets $\mathcal{O}_{Super}$ and $\mathcal{O}_{Sub}$. Assuming deterministic execution, sequential tests have exactly one output sequence. In contrast, concurrent tests can have multiple output sequences due to scheduling non-determinism. If there exists a sequence of return values that occurs only with Sub, the oracle reports a warning because Sub is an output-diverging substitute for $Super$. The analysis explores all possible interleavings of concurrent tests to avoid missing an output sequence (Section 3.4.4).

To compare output sequences of test executions, the oracle transforms the return values into an execution-independent format. For primitive values and Strings, the output sequence contains the actual return value. For reference values, the oracle stores whether the value is `null` or not. For example, the output sequence of executing the sequential test in Figure 3.1 with an instance of `TreeMap` is the following (values are separated with "—"):

null — non-null ref. value — int:0

Executing the concurrent test in Figure 3.2 with an instance of `Properties` gives two output sequences:

null — String:"b"

null — null

Alternatively to this encoding of return values, the output oracle could compare reference values with their `equals()` method, compare the String representations obtained via `toString()`, or compare the hash codes. Unfortunately, with these approaches objects often appear to be different even though they are semantically the same. The reason is that the default implementations of `equals()`, `toString()`,

Algorithm 4 Crash oracle. Checks whether Sub is a crashing substitute for $Super$.

```
Input: Generic test $t$ for classes $Super$ and $Sub$
Output: Warning about an unsafe substitute, or nothing
1: $\mathcal{F}_{Sub} \leftarrow execute'(t_{Sub})$
2: if $\mathcal{F}_{Sub} \neq \emptyset$ then
3:     $\mathcal{F}_{Super} \leftarrow execute'(t_{Super})$
4:     if $\mathcal{F}_{Super} = \emptyset$ then
5:         $reportWarning(t, \mathcal{F}_{Sub})$
```

and `hashCode()` refer to object identity. As a result, many of the warnings reported by the output oracle would be spurious because the difference in output is not a bug.

To avoid false warnings caused by calls to `equals()`, `toString()`, and `hashCode()` in the generated tests, the oracle ignores return values of those methods. With this refinement, the output sequence for Figure 3.1 with an instance of `TreeMap` is not the output sequence above but the following:

null — non-null ref. value — ignored

The output oracle compares the visible behavior of *Super* and *Sub* in a strict way. In practice, subclasses often change the return value of methods in ways that are considered to be in line with substitutability (we give examples in Section 3.6.4). Our experiments show that many of the warnings produced by the output oracle are benign differences that do not correspond to bugs in the analyzed classes (Section 3.6.2). In the following, we address this imprecision of the output oracle and present a precise oracle that reports only severe differences between a subclass and a superclass.

3.4.2 The Crash Oracle

The crash oracle checks whether a test exposes a subclass to be a crashing substitute for a superclass. Each warning reported by the crash oracle shows how a usage of the subclass can lead to an uncaught exception or a deadlock in a situation where using the superclass works fine. Since program crashes are certainly undesired behavior, each of the reported warnings points to a severe substitutability problem.

Algorithm 4 summarizes the crash oracle. Given a generic test t, the crash oracle first executes it with *Sub*'s constructor. The $execute'()$ function returns the set of failures observed while executing the test. A failure is either an exception or a deadlock. If executing the test with *Sub*'s constructor leads to one or more failures, *Sub* may be implemented incorrectly or the generated test may use the class incorrectly. For example, a generated test may pass an argument that causes an `IllegalArgumentException`. To check whether the observed failures are part of the expected behavior of *Sub*, the oracle executes the test with *Super*'s constructor. If *Super* also causes a failure, the oracle does not report a warning because the behavior of the subclass does not diverge from the behavior of the superclass. If, however, the set of failures from *Super* is empty, then the oracle reports a warning because *Sub* leads to a crash that is not possible with *Super*.

3.4.3 Distinguishing Concurrent from Sequential Problems

We refine both Algorithms 3 and 4 to distinguish problems that occur only in concurrent tests from problems that also appear sequentially. This refinement matters when our analysis checks for concurrent substitutability problems and when it should not report sequential problems.

Before reporting a warning for a concurrent test, the output oracle checks whether the output sequences $\mathcal{O}_{SubOnly}$ are also possible when linearizing the test into a single thread. A linearization of a concurrent test is a sequence of calls that contains all calls from the test while preserving the partial order of calls from the test. Since we generate concurrent tests with two suffixes that each contain a single call, there are exactly two linearizations for each concurrent test: One where the call from thread 1 comes first and one where the call from thread 2 comes first. If all output sequences in $\mathcal{O}_{SubOnly}$ also occur with linearizations of the concurrent test, then the substitutability problem is sequential and not specific to concurrently using Sub and $Super$.

The crash oracle executes the linearizations of a concurrent test before executing t_{Sub}. If the execution of at least one linearization fails, then the oracle does not explore the concurrent execution of the test and instead treats the failing linearization like a sequential test obtained from the test generator.

3.4.4 Exploring Executions

The $execute()$ function in Algorithm 3 and the $execute'()$ function in Algorithm 4 explore all possible executions of a test. $execute()$ returns the set of output sequences produced by these executions, and $execute'()$ returns the set of failures observed during the executions. For a sequential test, the execution functions simply run the test in a single thread of execution. We assume sequential executions to be deterministic. Sequential tests that do not behave deterministically, for example, because their behavior depends on the current system time, can be made deterministic by controlling the execution environment [122].

For concurrent tests, the execution functions use the stateful software model checker Java PathFinder (JPF) [158]. JPF systematically explores all possible interleavings of the concurrent suffixes of the test and therefore finds all possible behaviors (that is, output sequences or failures) of a concurrent test despite the nondeterminism introduced through scheduling. Exploring all possible executions is crucial to ensure that each warning reported by the superclass oracle is indeed a substitutability problem. If the set of behaviors from executing a test with $Super$'s constructor were incomplete, the oracle could report a warning even though the behavior observed with Sub is possible with $Super$. To avoid this problem, the oracle explores all possible executions. Checking all interleavings of concurrent calls faces the problem of combinatorial explosion. By focusing on two concurrent suffixes that each have a single call, we ensure that the oracle terminates in reasonable time (Section 3.6.7).

3.4.5 Examples

Figures 3.1 and 3.2 are two examples of problems detected by the crash oracle. In both examples, executing the test with the superclass constructor succeeds, while executing it with the subclass constructor leads to an exception. A client that calls methods in the same way as in the generated tests will be surprised about a program crash because the superclass does not show this behavior. Section 3.6.3 reports examples of warnings from the output oracle.

3.5 Implementation

We implemented the analysis into a fully automatic testing tool for Java classes. The implementation takes source code or byte code as input and reports unsafe subclasses as output. We implemented a helper tool to find all superclass-subclass pairs in particular packages and to generate constructor mappings as described in Section 3.3. Together with this helper tool, the implementation offers a push button technique to analyze entire class libraries with little effort.

To implement the superclass oracle we build upon JPF (version r615) for exploring concurrent executions. JPF is a mature and well-tested research tool but has some limitations, for example, when analyzing native code. Another problem is that even exploring only two concurrent threads that each have a single method call can take a significant amount of time. We deal with these problems by filtering the errors raised by JPF and by canceling JPF after a configurable timeout. If JPF crashes or if it cannot analyze a concurrent test within the timeout, we consider the test to be inconclusive. The superclass oracle does not report any warnings for inconclusive tests. That is, we risk missing some unsafe substitutes but avoid false warnings caused by JPF limitations. For our experiments, we set the timeout to ten seconds per test, which is sufficient for 96% of all concurrent tests.

The test generator creates tests with a configurable number of calls. For sequential tests, we set the maximum number of OUT calls to five after instantiating the OUT. For concurrent tests, the prefix contains at most five calls to the OUT.

3.6 Evaluation

We evaluate our approach with well-tested and widely used Java classes. Our main results are:

- Many classes are crashing substitutes (43/145=30%). All these unsafeties correspond to bugs that should be fixed because they may cause exceptions and deadlocks in clients.
- Even more classes are output-diverging substitutes (61/145=42%). We classify most of them (57/61) as false positives.
- Developers care about crashing substitutes. At least ten bugs found by the analysis have been fixed by now.

3.6.1 Experimental Setup

We apply the analysis to sequentially and concurrently used classes (Table 3.1). As sequentially used classes, we consider classes from three popular Java libraries: The Apache Commons Collections library, the XML processing library dom4j, and the PDF editing library iText. We select libraries, instead of closed programs, because every usage simulated by the test generator may happen in some client program. Table 3.1 shows the number of analyzed superclass-subclass pairs. We consider all pairs

Table 3.1: Summary of results. False positive are either due to acceptable differences (AD) or due to incorrect constructor mappings (ICM).

Subject	Seq./	Class	Crash oracle				Output oracle			
	Conc.	pairs	Warn.	Bugs	False pos.		Warn.	Bugs	False pos.	
					AD	ICM			AD	ICM
Commons Coll. 3.2.1	Seq.	12	5	5	0	0	3	2	0	1
dom4j 2.0.0-alpha-2	Seq.	46	12	12	0	0	26	0	24	2
iText 5.2.0	Seq.	58	21	21	0	0	30	1	9	20
Qualitas 20101126r	Conc.	29	5	5	0	0	2	1	1	0
Sum		145	43	43	0	0	61	4	34	23

of public and concrete classes that are in a direct or indirect subclass relationship, excluding the superclass `Object` and classes without public constructors—in total, 116 class pairs. As auxiliary classes for generating tests, we pass common classes from the Java standard library and all classes from the respective library. The test generator automatically selects those classes that provide required arguments.

As concurrently used classes, we consider subclasses of 30 classes that are known to be thread-safe and that are part of the Java standard library (Table A.5). We consider all 323,107 classes included in the Qualitas Corpus [150] and select a class if it extends one of the 30 thread-safe classes, if it is public and concrete, and if it has a public constructor. This selection yields 29 subclasses of thread-safe classes to analyze.

We analyze each class pair until a maximum number of tests has been generated: 500 tests for sequentially used classes and 3,000 tests for concurrently used classes. We set a larger limit for the latter because a single concurrent test checks only two methods against each other. To reach the maximum number of tests, we run the analysis with different random seeds, where the number of tests generated per seed increases with the number of seeds used. All experiments are done on an eight-core machine with 3GHz Intel Xeon processors and 8GB memory, running 32-bit Ubuntu Linux and the Java Hotspot VM version 1.6.0_27, giving 2GB memory to the VM.

We manually inspect all warnings reported by the analysis and classify them as bugs or false positives. A warning is a bug if the subclass behavior should be changed to avoid unexpected behavior in client programs, and it is a false positive otherwise. We distinguish between two kinds of false positives:

- *Acceptable difference*. Differences in behavior that we deem to be acceptable, for example, methods where clients expect that the outcome depends on the runtime type of the receiver.
- *Incorrect constructor mapping*. Generic tests where the superclass and subclass constructors are not semantically equivalent, that is, the heuristic constructor mapping (Section 3.3) is incorrect.

```
ArrayList l = new ArrayList() OR new Chapter();
Set empty = new HashSet();
boolean b = l.addAll(empty);
```

```
Result:
b==false if l is ArrayList
b==true  if l is Chapter
```

Figure 3.4: An output-diverging substitute that we classify as a bug.

3.6.2 Overview of Unsafe Substitutes Found

Table 3.1 shows for both the crash oracle and the output oracle how many warnings the analysis finds and how we classify them. For both oracles, a large percentage of the analyzed subclasses are unsafe substitutes: 30% are crashing substitutes and 42% are output-diverging substitutes. For the crash oracle, all reported warnings correspond to bugs, which is not surprising because the crash oracle focuses on certainly undesired behavior. In contrast, for the output oracle, most reported warnings are false positives. Two of the four bugs detected by the output oracle are also found by the crash oracle because the diverging subclass behavior manifests both through different output and through a crash. Table B.1 lists details about all warnings reported by the crash oracle.

Which oracle should developers use? We recommend the crash oracle as a default because it finds 96% of all detected bugs and because it does not report false positives. In contrast, the output oracle has a high false positive rate and finds few additional bugs. Yet, if developers know that the visible behavior of a particular subclass should be exactly the same as that of the superclass, then the output oracle can be used to test this property.

3.6.3 Examples of Unsafe Substitutes

Figures 3.1 and 3.2 are bugs found by the crash oracle. Figure 3.4 shows a bug found exclusively by the output oracle. `Chapter`, a subclass of `ArrayList`, modifies the meaning of the return value of `addAll()` in a subtle way: While the superclass returns whether the content of the list has changed, the subclass returns whether the operation was successful. The difference matters when an empty collection is passed to `addAll()`, as illustrated in the figure. We reported this bug and the developers fixed it.[5]

Figure 3.5 is a false positive reported by the output oracle. The subclass `DefaultNamespace` adds a functionality, support for a parent relationship, to its superclass `Namespace`. Both classes provide a method `supportsParent()` to check in a reflection-like manner whether an object supports this functionality. This behavioral difference is documented and should not surprise clients because the only purpose of the `supportsParent()` method is to find out whether an object supports the functionality, which is know to depend on the runtime type of the object.

[5]See issue 3548434 in the iText bug tracker.

```
Namespace ns = new Namespace("a", "b") OR new DefaultNamespace("a", "b");
boolean b = ns.supportsParent();
```

Result:
`b==false` if `ns` is `Namespace`
`b==true` if `ns` is `DefaultNamespace`

Figure 3.5: An output-diverging substitute that we classify as a false positive.

3.6.4 Root Causes for Unsafe Substitutes

The large percentage of unsafe substitutes raises the question why subclasses extend their superclasses in an unsafe way. For unsafe substitutes that correspond to bugs, the most common root causes are:

- *Stronger precondition for method arguments*. An overriding method imposes a stronger precondition for method arguments than the overridden method. For example, some overriding methods expect only a subset of all subtypes of the declared parameter type as actual parameters, and passing arguments of other compatible types raises a `ClassCastException`.
- *Stronger precondition for method receiver*. An overriding method has a stronger precondition on the state of the receiver object than the overridden method. For example, some overriding methods access fields that are not used by the overridden methods.
- *Removed methods*. A subclass explicitly "invalidates" a method of the superclass by throwing an `UnsupportedOperationException` and thereby breaks the type system's safety guarantee that each method call is understood.
- *Propagated unsafety*. A subclass extends a class that itself is an unsafe substitute, and the unsafety propagates down the inheritance hierarchy. Given a class A with an unsafe substitute A', a subclass A'' of A' is also an unsafe substitute for A, unless A'' fixes the problem introduced by A'.

  For example, some unsafe substitutes in iText are the result of extending `Properties`, which is an unsafe substitute for `Hashtable`. The problem is that `Properties` is assumed to map `String`s to each other but extends `Hashtable<Object,Object>`, from which it inherits methods to put arbitrary objects into the table. Some methods of `Properties`, such as `propertyNames()`, cast entries to `String`, which fails if the table contains entries of other types. Although this problem of `Properties` is documented, it not only may surprise clients of `Hashtable` that refer to a `Properties` instance, but it also affects subclasses of `Properties`.
- *Missing synchronization*. A subclass of a thread-safe class overrides a method without providing the synchronization guarantees that the overridden method provides. For example, some methods override a synchronized method and remove the `synchronized` keyword without ensuring synchronization in another way.

Table 3.2: Bugs reported to developers.

Issue	Status
Castor 2729	Fixed within a day (reported by others)
Commons Collections 394	Fixed by removing the class
Commons Collections 423	Fixed by removing the class
Commons Collections 424	Acknowledged, working on it
dom4j 3547635	Reported
dom4j 3547784	Reported
iText 3547811	Fixed within a day
iText 3547812	Fixed within a day
iText 3548434	Fixed within a day
JBoss Common 126	Fixed within a week
OpenJPA 2243	Fixed within a day

One subclass, `NonSynchronizedVector` from Eclipse, obviously removes synchronization on purpose. While this may increase performance, it poses a significant risk at clients of `Vector` because such clients may unconsciously refer to an instance of `NonSynchronizedVector`.

For unsafe substitutes that we classify as false positives, the most common root causes are:

- *Ad hoc reflection*. A method provides hints about the runtime type of an object or about the functionalities supported by this type (for example, see Figure 3.5). These methods are an ad hoc form of reflection, and clients calling them should be aware that superclass behavior and subclass behavior may differ.
- *String representations*. A method returns a String representation of the receiver object and the String contains type-specific information. The output oracle filters calls to `toString()` to avoid more false positives of this kind, but it does not handle application-specific methods that return String representations.

3.6.5 Failures Observed by the Crash Oracle

The crash oracle depends on signs of certainly undesired behavior, such as exceptions and deadlocks. This dependence may raise the question to what extent our results depend on a defensive programming style, where illegal state and illegal arguments are made explicit by throwing exceptions. We categorize the failures that expose crashing substitutes by whether they are exceptions raised in the analyzed code base or not. Failures not triggered in the analyzed code base are exceptions raised in the Java standard library or in the virtual machine, or deadlocks. 84% of the failures that expose the crashing substitutes in Table 3.1 are not triggered in the analyzed code base, that is, independent of a defensive programming style.

3.6.6 Feedback from Developers

Do developers really care about substitutability problems? To answer this question, we report a subset of the bugs found by the analysis to the developers of the respective projects (Table 3.2). By the time of this writing, seven of ten reported bugs have been fixed as a reaction to our reports. Another bug has been reported and fixed independently of us. Moreover, at least two other bugs found by the analysis (not listed in Table 3.2) have been fixed in iText 5.3.0 independently of us, but we could not find a corresponding entry in the project's bug tracker. Many of the remaining unreported bugs are in deprecated classes that are likely to be replaced soon. Overall, the feedback from developers suggests that they care about substitutability problems, which confirms our expectation that unsafe substitutes that can surprise clients are not desired and should be fixed.

3.6.7 Performance

To measure performance, we test each class pair until either the maximum number of tests has been generated or until the subclass has been found to be a crashing substitute. For sequentially used classes, our prototype implementation takes on average 41 seconds to find crashing substitutes and 19 seconds to find output-diverging substitutes. For concurrently used classes, it takes on average 21 minutes to find crashing substitutes and 108 minutes to find output-diverging substitutes. The reason why concurrent testing takes longer is that the analysis explores all interleavings of concurrent tests.

3.6.8 Threats to Validity

There a several threats to the validity of this evaluation. First, the set of classes we analyze may not be representative for any larger set of Java classes. In particular, we focus the evaluation on libraries and cannot draw conclusions about the effectiveness of the approach for non-library classes. Second, the parameters of the test generator, such as *maxStateChangerTries*, may influence the results. We set these parameters based on our intuition and the results from initial experiments, and therefore, we cannot guarantee them to be optimal in any sense. Third, the set of auxiliary classes provided to the test generator influences the ability of the analysis to find unsafe substitutes, and we may accidentally have selected auxiliary classes that bias our results.[6] Forth, the selection of problems we report to developers may not be representative for all problems that the analysis detects. That is, we cannot conclude which percentage of all detected problems is considered relevant by developers. Fifth, the maximum number of generated tests influences the number of reported warnings. A different stopping criterion may give different results. Finally, the assessment of whether a warning is a true positive relies on our understanding of substitutability. Other developers may classify warnings differently.

3.7 Support for the Thesis

The presented analysis supports our thesis that automatic program analysis allows for precisely detecting programming errors with little effort. We restrict this claim and the following discussion to the crash oracle, which we recommend to use as the only oracle by default.

[6]The second and the third threat to validity is shared with the analysis in Chapter 2 because the test generation approaches are similar.

3.7.1 Automation

Given a subclass and a superclass, the analysis fully automatically tests whether the first is a safe substitute for the latter. This degree of automation is enabled by generating generic tests that exercise these classes, by heuristically creating constructor mappings, and by leveraging the superclass as an executable specification of correct subclass behavior.

3.7.2 Precision

During our evaluation with 145 pairs of classes, the analysis detects 43 substitutability bugs without reporting a single false positive. In principle, false positives can occur because the analysis relies on a heuristic mapping from subclass constructors to superclass constructors. If two constructors map to each other but do not produce semantically equivalent objects, the warnings of the analysis may be spurious. We did not encounter this situation with the crash oracle, but give an example of such a spurious warning from the output oracle in Section 3.6.3.

To guarantee the precision of the analysis, developers can refine the heuristically generated constructor mappings manually. That is, developers can trade automation for precision. Based on our experimental results, we recommend to use automatically generated constructor mappings because they yield high precision in practice.

3.7.3 Effort

The approach requires very low human and acceptably low computational effort to find bugs. Human effort is low because the analysis requires no input except for the software to test and—at least in practice—produces no output but true positives. The computational effort is low for sequentially used classes (41 seconds on average to find a bug). For concurrently used classes, the analysis takes longer (21 minutes on average to find a bug) because it explores concurrent executions exhaustively. We consider the performance for both sequentially and concurrently used classes to be acceptable for a deployment in practice because the analysis runs without human interaction and because the analysis detects bugs that traditional testing did not reveal. Parallelizing the analysis, for example, by concurrently generating tests with different seeds, is an easy way to further reduce the computational effort of our prototype implementation.

3.8 Limitations and Future Work

Directions of future work include:

- *More precise output oracle*. The current output oracle reports mostly false positives but also reveals some bugs not found by the crash oracle. Improving the precision of the output oracle seems an interesting challenge for future research. For example, one could filter methods that give the same result for all instances of a class to avoid false positives due to ad hoc reflection, such as the warning shown in Figure 3.5.

- *Classes of closed programs*. The current analysis targets library classes, which can be used by clients in arbitrary ways. For classes of closed programs, randomly generated tests may not always represent realistic usages. Future work may adapt our approach to classes of closed programs, for example, by mining the usages of a class within a program and by guiding the test generator to create similar usages.

API Protocol Violations 4

Most programs reuse existing software components. Reuse saves development time because it allows for leveraging well-tested implementations of common functionality with relatively little effort. For example, most non-trivial Java programs use parts of the Java standard library, such as the collections in `java.util` or the input/output facilities in `java.io`. Reusable software components are accessed through application programming interfaces (APIs), that is, well-defined ways to access data and functionality. In Java software, an API consists of a set of classes and interfaces that provide methods and fields with well-defined signatures.

While reuse reduces development time, it also entails an additional burden for developers: Many APIs are hard to learn. A study with 83 industrial developers reveals several obstacles for API learning, including the lack of adequate examples, problems to understand the API's structural design, and incomplete documentation of common APIs usage patterns [136]. A particular obstacle to correctly using an API is that many APIs have *API usage protocols*. Such protocols impose constraints on the order in which API clients are supposed to call API methods. For example, an API may specify that a method should only be called after calling another method, or that calling a method is illegal while calling a set of other methods. Such protocols are common in practice: Beckman et al. show that at least 7.2% of all types in a large corpus of open source Java programs impose protocols, and that at least 13% of all classes are clients of types that impose protocols [12].

API protocols are burdensome and cause bugs in real software. A study of developer forums of two popular APIs shows that developers have various questions related to protocols and that solving protocol-related problems can take several days even with expert help [77]. Another study investigates bug fixing patterns and proposes two categories of bug fixes that are strongly related to protocols: "addition of operations in an operation sequence of method calls to an object" and "removal of operations from an operation sequence of method calls to an object" [119]. Together, these two categories contain thousands of bug fixes in the analyzed projects and account for 6.7% of all classified bug fixes. These number suggests that protocol bugs are common and that developers care enough about them to fix them.

Unfortunately, finding bugs due to violations of API protocols is hard. Existing tools used by practitioners, such as FindBugs [75] and PMD [2], address some protocol bugs. However, these bugs are only a small subset of all possible protocol bugs because the respective tools must contain a hand-crafted checking rule for each kind of bug. The main reason why finding protocol bugs is hard is that most protocols are not formally specified. Instead, protocols are documented informally in Javadoc comments or even not documented at all. This lack of specification not only makes understanding and respecting protocols difficult for developers, but it also poses a problem at program analyses that aim at checking whether a program violates protocols.

Can automatic and precise bug detection help developers to find protocol bugs? This chapter presents a positive answer to this question. We present a fully automatic, dynamic analysis that searches for violations of API protocols. To find protocol violations despite the lack of formal protocol specifications, the analysis infers protocols from usages of the API and then checks a program against the inferred protocols. To achieve full automation, the analysis leverages generated tests as a driver for dynamically inferring and checking protocols. We implement and evaluate the approach and show that automatic and precise bug detection helps in finding protocol bugs with little effort.

The following section defines API usage protocols and illustrates them with an example. Sections 4.3 to 4.6 and Section 4.7 present the approach and a heuristic optimization of it, respectively. Section 4.8 evaluates the approach. Finally, Sections 4.9 and 4.10 discuss how the approach fits the thesis of this work and outline directions of future work.

4.1 API Usage Protocols

Before presenting the approach, we give a definition of protocols and illustrate it with examples.

Definition 9 (API usage protocol). *An API usage protocol $P = (M, \mathcal{P})$ consists of a deterministic finite state machine M and a finite set $\mathcal{P}$ of typed protocol parameters. M is a tuple $(\mathcal{S}, \Sigma, \delta, s_0, \mathcal{S}_f)$ of states $\mathcal{S}$, the alphabet Σ, transitions δ, the initial state $s_0 \in \mathcal{S}$, and final states $\mathcal{S}_f \subseteq S$. A transition is a triple from $\mathcal{S} \times \Sigma \times \mathcal{S}$, which defines the source state, the label, and the destination state of the transition. The alphabet Σ consists of method signatures that are annotated with protocol parameters naming the receiver, and optionally, the method parameters and the return value.*

For example, Figure 4.1 is a protocol for using a `Stack`. After instantiating the stack, a client pushes an element onto it and reaches state 3. At this state, a client can push or pop elements an arbitrary number of times until the client calls `remove-`

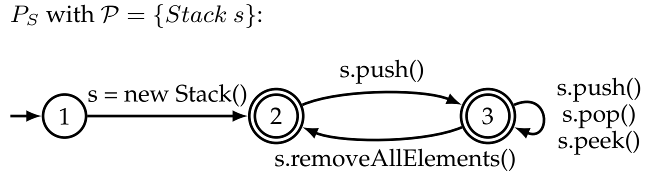


Figure 4.1: Inferred protocol for `java.util.Stack`. A label $p = m()$ means that the object returned by $m()$ is bound to protocol parameter p.

P_{CI} with $\mathcal{P} = \{Collection\ c, Iterator\ i\}$:

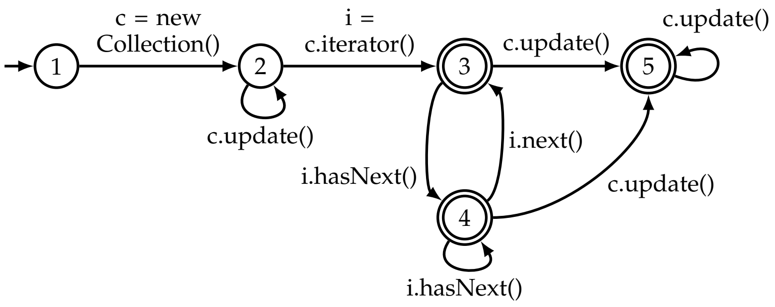


Figure 4.2: API usage protocol describing how to use a collection and an iterator. The call $c.update()$ summarizes calls that may change the collection's content, for example, $c.add()$ or $c.remove()$.

`AllElements()`, which brings the protocol back to state 2. The protocol is a regular language approximation of the full stack specification, which requires balancing calls to `push()` and `pop()`. Despite this approximation, the protocol can reveal illegal usages of stacks. For example, a client that calls `pop()` on an empty stack violates the protocol.

As a second example, Figure 4.2 shows a protocol that involves two objects. The protocol describes how to use a collection (`java.util.Collection`) together with an iterator (`java.util.Iterator`). The states in the protocol represent the combined states of the involved objects, that is, the combined state of a collection c and an iterator i. Transitions correspond to method calls, where one of the involved objects is the receiver. The protocol specifies several constraints on using collections and iterators:

- At first, a client creates a collection and then the iterator. That is, it is not possible to create an iterator first and to assign it to a collection afterwards.
- There are three main phases of using a collection and an iterator: First, the collection is created and updated an arbitrary number of times, for example, by adding and removing elements (states 1 and 2). Second, the iterator is created and used to iterate through the collection elements (states 3 and 4). Finally, the collection may be updated again, which invalidates the iterator. That is, once the protocol reaches state 5, the iterator cannot be used anymore.
- The usage can be stopped at any time after creating the iterator, as indicated by states 3 to 5 being final states. Of course, it is also possible to use a collection without any iterator. This collection-only usage can be specified in a protocol involving only a collection and no iterator.

  Other protocols may require an API client to call particular methods to finalize a usage, for example, to release resources to the operating system.

- Each call to the iterator's $next()$ method must be preceded by a call to $hasNext()$ to check whether there is a further element.

Violations of the protocol can lead to unexpected behavior and even program crashes:

- Using an iterator that is invalid because the underlying collection has been updated may raise a `ConcurrentModificationException` or lead to undefined behavior.
- Calling `next()` on an iterator that has no further elements raises an `Illegal-StateException`.

4.2 Overview of the Approach

APIs have usage protocols and, as the examples in Figures 4.1 and 4.2 show, violating these protocols leads to undesired behavior. Which part of a program is responsible for respecting an API usage protocol and how can an automatic and precise bug detection approach help avoiding violations of protocols? In this section, we outline our answers to these questions. The remaining sections fill in the details of the approach and evaluate its effectiveness.

4.2.1 Problem Definition

To understand the problem we address, consider a part of a program as shown in Figure 4.3. A class, called `X`, uses an API class (or a set of related API classes) that impose(s) an API usage protocol. `X` itself is used by another class in the program, called the client `C`. The fact that `X` uses the API is an implementation detail of `X`, and `C` may not know about it. Therefore, class `X` is responsible for following the API protocol. In particular, `X` must protect `C` from unintentionally violating the protocol.

Many APIs throw exceptions when a protocol is violated. Often, these exceptions are declared by the methods that may throw them. For example, `Stack.pop()` declares in its Javadoc to throw an `EmptyStackException` when the method is called on an empty stack. Making explicit that violating a protocol leads to an exception helps users of an API to avoid protocol violations.

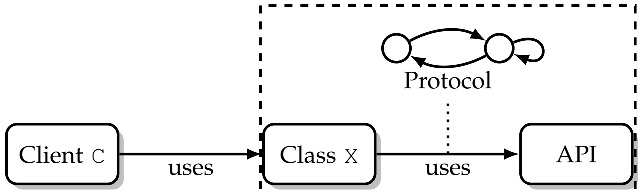


Figure 4.3: For safe API usage, `X` must ensure to follow the API protocol. In particular, no exception caused by an unexpected protocol violation should leave the dashed box.

```
class X {
  private Stack s = new Stack();
  public String report() {
    return get().toString();
  }
  private Object get() {
    s.peek();
  }
  public void fill() {
    s.push(..);
  }
}
```

Figure 4.4: Unsafe API usage that can violate the `Stack` protocol and an execution trace exposing a protocol violation.

What does a declared exception mean for the situation in Figure 4.3? Class `X` has three options to use the API in way that protects the client `C` from API protocol violations:

- `X` follows the API protocol and does not trigger exceptions due to protocol violations.
- `X` may violate the protocol but catches the resulting exceptions to avoid surprising `C`.
- `X` may violate the protocol and explicitly propagates the resulting exceptions by declaring them in its Javadoc or its method signature. In this case, `X` itself has a protocol and makes it as explicit to `C` as the underlying API usage protocol is to `X`.

We call a usage of an API that matches one of the three options a *safe API usage*, and we call it an *unsafe API usage* otherwise. Programmers of a class `X` should avoid unsafe API usages to avoid surprising client classes, such as `C`. Since the usage of an API is an implementation detail of `X`, `C` cannot foresee exceptions caused by protocol violations in `X`.

As a motivating example, consider a simplified version of a problem our analysis found in Xalan, a program shipped with the DaCapo 2006-10-MR2 benchmarks (Figure 4.4). A class `X`[1] offers a public method `report()`. Calling `report()` on a fresh instance of `X` gives an `EmptyStackException`, which is not declared in the public interface of `report()`. The problem is that `X` uses the `Stack` API in an unsafe way that exposes users of `X` to exceptions caused by violations of `Stack`'s API protocol. A user of `X` cannot foresee that `X` may crash the program, unless the user inspects `X`'s implementation, which however, contradicts the idea of modular, object-oriented programming.

[1]The original class in Xalan is called `StylesheetHandler`. We edited it for conciseness.

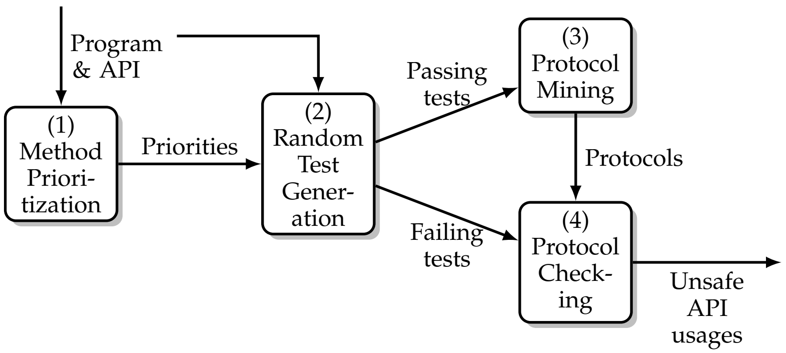


Figure 4.5: Overview of the approach.

4.2.2 Our Approach

Developers can find unsafe API usages with an automatic and precise bug detection approach. The approach builds upon three existing techniques that—on their own—do not fully address the problem of revealing violations of API usage protocols. First, we build upon *automatic test generation* [36, 117, 30]. Even though state of the art test generators achieve good coverage, their usefulness is bounded by the test oracle that decides when to report a warning to developers. For example, the Randoop test generator [117] comes with a small set of manually specified, generic test oracles that do not detect all bugs triggered by the generated tests.

Second, we build upon *dynamic protocol mining*, which infers API usage protocols from execution traces of a program that uses an API. The usefulness of a dynamic protocol miner is bounded by the executions it analyzes. A protocol miner can infer a protocol only if there is input that exercises paths of an API client that expose the protocol.

Third, we build upon *dynamic protocol checking* [6, 105, 27], which verifies whether a program execution violates a given protocol. Similar to a dynamic protocol miner, the usefulness of a dynamic protocol checker is bounded by the executions it analyzes. The checker can report a protocol violation only if some input triggers the illegal behavior. Furthermore, protocol checking requires specifications of the protocols to check.

Can test generation, protocol mining, and protocol checking help finding the problem in Figure 4.4? A test generator can produce tests that cover many paths through `C`, including a path that triggers the `EmptyStackException`. Unfortunately, an exception alone is an unreliable indicator for a bug: Generated tests produce many exceptions due to illegal usage, for example, by passing illegal method arguments. A protocol miner can infer the protocol for using `Stack`s but requires input that drives clients of the `Stack` class. Finally, a runtime protocol checker can detect that `C` violates the `Stack` protocol but requires the protocol and input that drives `C` into the protocol violation.

Our approach reveals the unsafe API usages by combining test generation, protocol mining, and protocol checking as illustrated in Figure 4.5. Given a program and an API as input, we use a random test generator (2) to generate tests for the program. The generated tests can be classified into two groups: *failing tests* that lead the program to an exception, and *passing tests* that execute without any obvious error. We analyze executions of passing tests with a dynamic protocol miner (3), which produces protocols describing likely specifications of how to use API classes (Section 4.4). Then, we analyze executions of failing tests with a dynamic protocol checker (4) (Section 4.5). This last step filters the various failing tests by selecting those that fail because the program violates a protocol, removing all tests that fail for other reasons, such as illegal, generated input (Section 4.6). Finally, the protocol violations are reported to the developer as warnings about unsafe API usages.

Figure 4.5 also shows *method prioritization* (1), a static analysis that heuristically computes how relevant each of the program's methods is for triggering calls into the API. Method prioritization is an optional optimization and can be ignored for the description of our general approach. Section 4.7 explains how method prioritization makes our technique even more useful in practice.

4.3 Random Test Generation

There are various approaches to automatically generate tests (see Section 7.5 for a detailed discussion). Our approach for finding protocol bugs builds upon feedback-directed, random test generation, implemented in Randoop [117]. We use Randoop because it requires no input except the program under test, scales well to large programs, and generates tests with high coverage.

Randoop randomly selects methods and chains them into call sequences. If executing a call sequence succeeds (defined below), the sequence is used to construct further call sequences until a maximum sequence length is reached. To report warnings to developers, Randoop checks generated call sequences against a built-in set of generic test oracles (called "contracts" in [117]). For example, Randoop reports a warning if `o.equals(o)` returns false or if a method throws a `NullPointer-Exception` even though all input arguments are non-null. A sequence is said to fail if it raises an exception or if it violates any of the built-in checks, and the sequence is said to succeed otherwise.

In a typical Randoop usage scenario, a user is mostly interested in failures of the built-in checks. Instead, we use Randoop by considering both failing and succeeding sequences. We configure Randoop to report succeeding sequences as *succeeding tests* and failing sequence as *failing tests*. Unless mentioned otherwise, we leave all Randoop parameters at their default values, that is, as described in [117].

4.4 Protocol Mining

To obtain specifications of how to use an API, we use a dynamic protocol miner. The miner analyses a running program and produces API usage protocols that summarize the API usages observed during the program's execution. In this section, we present the protocol miner and how we fit it into an automatic and precise bug detection approach. Evaluating the mining approach is an interesting topic on its own. We address this topic with an evaluation framework for protocol miners [124], which however, is beyond the scope of this thesis. The protocol miner has applications beyond what we present here. For example, the protocols that it infers can be fed into a

```
class Bar {
  void m(List<Foo> list, Reader r) {
    Iterator<Foo> iter = list.iterator();
    while (iter.hasNext()) {
      Foo foo = iter.next();
      ...
    }

    BufferedReader br = new BufferedReader(r);
    String line = br.readLine();
    while (line != null) {
      list.add(toFoo(line));
      line = br.readLine();
    }
    br.close();
  }

  private Foo toFoo(String s) { ... }
}
```

Figure 4.6: Method `m()` iterates over a `List`, reads lines via a `BufferedReader` and adds them to the list.

static program checker [131]. However, applications of the protocol miner other than the one reported in this chapter are also beyond the scope of this thesis.

The protocol mining divides the task of mining protocols into three subtasks. First, we instrument a program so that it writes an execution trace into a file (Section 4.4.1). Second, the protocol miner splits the execution trace into subtraces that each contain a sequence of related calls (Section 4.4.2). Third, the protocol miner groups similar subtrace together and generates an FSM for each group of subtraces (Section 4.4.3).

4.4.1 Gathering Execution Traces

The first step of mining protocols from program executions is to log runtime events into an execution trace.

Definition 10 (Execution trace)**.** *An* execution trace *is a sequence of method calls caller $\rightarrow$ callee and method returns caller $\leftarrow$ callee that have been observed during the execution of a program. For each caller and callee, the trace contains the following information:*

- *Signature of the called method.*
- *Runtime type and identifier of the receiver.*

For each method call, the trace contains the runtime type and identifier of the arguments. For each method return, the trace contains the runtime type and identifier of the return value (if any).

```
... --> Bar{bar}.m()
  Bar{bar}.m() --> LinkedList{list}.iterator()
  Bar{bar}.m() <-- LinkedList{list}.iterator(): ListItr{iter}
  Bar{bar}.m() --> ListItr{iter}.hasNext()
  Bar{bar}.m() <-- ListItr{iter}.hasNext(): boolean{boolean1}
  Bar{bar}.m() --> ListItr{iter}.next()
  Bar{bar}.m() <-- ListItr{iter}.next(): Foo{foo1}
  ...
  Bar{bar}.m() --> ListItr{iter}.hasNext()
  Bar{bar}.m() <-- ListItr{iter}.hasNext(): boolean{boolean2}
  Bar{bar}.m() --> BufferedReader(Reader{r})
  Bar{bar}.m() <-- BufferedReader(Reader{r}): BufferedReader{br}
  Bar{bar}.m() --> BufferedReader{br}.readLine()
  Bar{bar}.m() <-- BufferedReader{br}.readLine(): String{line1}
  Bar{bar}.m() --> Bar{bar}.toFoo(String{line1})
    ...
  Bar{bar}.m() <-- Bar{bar}.toFoo(String{line1}): Foo{foo2}
  Bar{bar}.m() --> LinkedList{list}.add(Foo{foo2})
  Bar{bar}.m() <-- LinkedList{list}.add(Foo{foo2}): boolean{boolean3}
  Bar{bar}.m() --> BufferedReader{br}.readLine()
  Bar{bar}.m() <-- BufferedReader{br}.readLine(): String{line2}
  Bar{bar}.m() --> BufferedReader{br}.close()
  Bar{bar}.m() <-- BufferedReader{br}.close(): void
... <-- Bar{bar}.m(): void
```

Figure 4.7: Execution trace from calling method `m()` in Figure 4.6. The notation `Type{object}` identifies an object of a particular type.

As a running example, consider the Java source code in Figure 4.6. Executing method `m()` may produce the execution trace in Figure 4.7.

To gather execution traces, our implementation builds upon aspect-oriented programming and the AspectJ compiler [1]. An aspect with two pointcuts adds instructions before each method call and after each method return. The corresponding advices pass the runtime information to a logging component, which writes the method call and return events into log files. For multi-threaded programs, we create a separate log file for each thread. We instrument only classes that belong to the analyzed program and omit libraries and frameworks. In particular, we do not instrument the Java standard library.

4.4.2 Extracting Subtraces from Large Execution Traces

Given an execution trace, the protocol miner extracts subtraces that each contain a sequence of related calls. As a motivating example, consider the execution trace in Figure 4.7. The trace contains API method calls involving API types and calls to other methods in the program. The goal of extracting subtraces is to unravel these calls so that each subtrace contains semantically related calls. The main idea for creating such subtraces is to focus on a particular core object and to include only calls that are strongly related to this core object.

Definition 11 (Subtrace)**.** *A subtrace for a core object o includes the following calls:*

- *All calls to o.*
- *For each parameter p of a call to o:*
  - *The (constructor) call that returned p.*
  - *All calls on p before it is passed to o.*
- *For each return value r of a call to o:*
  - *All calls on r after its return from o.*

Figure 4.8 shows the subtraces that the protocol miner extracts for the running example. There are four subtraces, each focusing on a different core object. A single call in the execution trace can be part of multiple subtrace. For example, the calls to the `ListItr` are part of both the `LinkedList`'s subtrace and the `ListItr`'s subtrace.

Generic Filters and Transformations

Before the protocol miner further analyzes subtraces, it applies a set of generic filters and transformations. Each filter and transformation takes one subtrace as its input and produces a possibly empty set of subtraces. If a filter or transformation produces multiple subtraces, all subsequent filters and transformations are applied to each of them. The protocol miner applies the filters and transformations in the order in which they are described in the following.

Remove Clone Objects Subtraces often contain multiple objects so that on each of them the same sequence of method calls is invoked. We call such objects *cloned objects*. The reason for cloned objects is repeated behavior due to loops in the analyzed programs. An API usage protocol that contains multiple cloned objects is bloated with calls and objects that often are not necessary to specify the API usage. Therefore, the protocol miner removes all but one cloned object from a subtrace as follows. The protocol miner compares the sets of methods called on each receiver object with each other. For each set of receiver objects where the same set of methods is called on it, the analysis randomly selects one of the receiver objects and removes all calls where any of the other receiver objects is the receiver from the subtrace.

Maximum Number of Objects During our experiments, we found that most subtraces with a large number of objects are bloated with objects and calls that do not relate to each other. Yet, most documented API usage constraints involve a relatively small number of interacting objects. Therefore, the protocol miner removes all subtraces that involve more than a user-specified number of receiver objects. For our evaluation, we set the maximum number of receiver objects to three.

Generalizing Types Observed runtime behavior can often be generalized with respect to the involved types. As a motivating example, consider object `iter` from the running example. Its dynamic type is `ListItr`, but the methods called on the object, `hasNext()` and `next()`, are also available in `Iterator`, which is a supertype of `ListItr`. The protocol miner generalizes the behavior observed for the `ListItr` to `Iterator` to infer more general API usage protocols.

```
// core object: list
Bar{bar}.m() --> LinkedList{list}.iterator()
Bar{bar}.m() <-- LinkedList{list}.iterator(): ListItr{iter}
Bar{bar}.m() --> ListItr{iter}.hasNext()
Bar{bar}.m() <-- ListItr{iter}.hasNext(): boolean{boolean1}
Bar{bar}.m() --> ListItr{iter}.next()
Bar{bar}.m() <-- ListItr{iter}.next(): Foo{foo1}
Bar{bar}.m() --> ListItr{iter}.hasNext()
Bar{bar}.m() <-- ListItr{iter}.hasNext(): boolean{boolean2}
Bar{bar}.m() --> LinkedList{list}.add(Foo{foo2})
Bar{bar}.m() <-- LinkedList{list}.add(Foo{foo2}): boolean{boolean3}

// core object: iter
Bar{bar}.m() --> ListItr{iter}.hasNext()
Bar{bar}.m() <-- ListItr{iter}.hasNext(): boolean{boolean1}
Bar{bar}.m() --> ListItr{iter}.next()
Bar{bar}.m() <-- ListItr{iter}.next(): Foo{foo1}
Bar{bar}.m() --> ListItr{iter}.hasNext()
Bar{bar}.m() <-- ListItr{iter}.hasNext(): boolean{boolean2}

// core object: br
Bar{bar}.m() --> BufferedReader(Reader{r})
Bar{bar}.m() <-- BufferedReader(Reader{r}): BufferedReader{br}
Bar{bar}.m() --> BufferedReader{br}.readLine()
Bar{bar}.m() <-- BufferedReader{br}.readLine(): String{line1}
Bar{bar}.m() --> BufferedReader{br}.readLine()
Bar{bar}.m() <-- BufferedReader{br}.readLine(): String{line2}
Bar{bar}.m() --> BufferedReader{br}.close()
Bar{bar}.m() <-- BufferedReader{br}.close(): void

// core object: bar
... --> Bar{bar}.m()
Bar{bar}.m() --> Bar{bar}.toFoo(String{line1})
Bar{bar}.m() <-- Bar{bar}.toFoo(String{line1}): Foo{foo2}
... <-- Bar{bar}.m(): void
```

Figure 4.8: Subtraces for the execution trace in Figure 4.7.

The analysis generalizes a subtrace into a set of subtraces by generalizing the type of each receiver object in the original trace and by combining generalized types with each other. Let T be a subtrace and let $\mathcal{R}(T)$ denote the set of receiver objects in the subtrace. At first, the analysis computes for each receiver object the set of types that provide all methods called on the object, that is, all types that the object could potentially have. That is, for each $r \in \mathcal{R}$, we compute the set $\mathcal{G}(r)$ of generalized types so that for each $t_g \in \mathcal{G}$:

- t_g is equal to or a supertype of the type t of r in T
- t_g provides all methods called on r in T

Then, the analysis creates a new subtrace for each combination of generalized types of the original subtrace's receiver objects. That is, given a subtrace with n receiver objects $r_1, \ldots, r_n$, the analysis creates $|\mathcal{G}(r_1)| \times \cdots \times |\mathcal{G}(r_n)|$ subtraces. Despite this combinatorial approach, the number of resulting subtraces is manageable in practice because we bound the number of receiver objects in a subtrace and because the number of generalized types is typically small.

Focus on API Methods The execution trace contains all calls issued during the execution of a program, but not all of these calls are related to the analyzed API. For example, a subtraces of our running example contains calls to `Bar.m()` and `Bar.toFoo()`, which are not related to the API of the Java standard library. To focus on a particular API, a user can specify packages to focus on, and the protocol miner will keep only calls to method defined in these API packages.

It is important to apply this filter after generalizing types because some types may only appear as API types after generalizing them. For example, a program may have a class that implements the `java.util.Collection` interface. After generalizing types, calls to this class may appear as API calls and are therefore considered for further analysis.

Remove Calls to Ubiquitous Types Most Java programs intensively use a small set of types provided by the Java standard library. As these types are ubiquitous, calls to them are often intermingled with other API usages without being an essential part of the API usage. To address the problem of such ubiquitous types, the protocol miner removes calls to a user-specified set of types. As a default, the protocol miner removes all calls to `String`, `CharSequence`, and `Object`. Other ubiquitous types can be easily added if necessary.

Require at Least Two Methods After applying the above filters and transformations, some subtraces may contain calls to less than two methods. The purpose of API usage protocols is to specify how multiple API methods are used together. Therefore, the protocol miner removes all subtraces that involve less than two methods.

4.4.3 Generating Finite State Machines

The final step of the protocol miner is to create API usage protocols from subtraces. At first, the miner groups similar subtraces together. Then, it infers an FSM from each group of subtraces. To group subtraces, the protocol miner assigns all those subtraces to the same group that involve the same set of receiver types. For example, all subtraces involving a `Collection` and an `Iterator` belong to the same group.

To create an FSM, we must define how to abstract states and how to connect them with transitions. As a state abstraction, our protocol miner considers the last k distinct called methods within a subtrace and the set of bound protocol parameters. Each distinct pair of a list of k most recently called methods and a set of parameters has its own state. In the initial state, the list of called methods and the set of bound parameters are empty. The protocol miner connects states with transitions that indicate that, while being in the source state, a particular call has been observed to lead to the destination state.

Algorithm 5 summarizes our approach to generate an API usage protocol from a set $\mathcal{T}$ of subtraces. The algorithm iterates over all subtraces and visits each call

Algorithm 5 Generates an API usage protocol P from a set $\mathcal{T}$ of subtraces.

```
1: $\mathcal{S} \leftarrow \emptyset$; $\Sigma \leftarrow \emptyset$; $\delta \leftarrow \emptyset$; $s_0 \leftarrow$ new state; $\mathcal{S}_f \leftarrow \emptyset$
2: **for** $T \in \mathcal{T}$ **do**
3:     $current \leftarrow s_0$
4:     $lastK \leftarrow$ empty FIFO queue of size $k$
5:     $R \leftarrow receiversToParams(T)$        ▷ Maps receivers to protocol parameters
6:     $boundParams \leftarrow \emptyset$
7:     **for** $call \in T$ **do**
8:         **if** $retVal(call) \in domain(R)$ **then**
9:             $boundParams \leftarrow boundParams \cup \{retVal(call)\}$
10:        $label \leftarrow computeLabel(call, R)$        ▷ Adds new labels to $\Sigma$
11:        **if** $lastK.last \neq label$ **then**
12:            $lastK.add(label)$
13:        $next \leftarrow nextState(lastK, boundParams)$        ▷ Adds new states to $\mathcal{S}$
14:        $\delta \leftarrow \delta \cup \{(current, label, next)\}$
15:        $current \leftarrow next$
16:    $S_f \leftarrow S_f \cup \{current\}$
17: $P \leftarrow (range(R), (\mathcal{S}, \Sigma, \delta, s_0, \mathcal{S}_f))$
```

in a subtrace once. The $receiversToParams()$ function (line 5) maps each receiver object in the subtrace T to a protocol parameter. Since all subtraces in $\mathcal{T}$ have the same multi-set of receiver types, the range of the returned map is the same for all subtraces in $\mathcal{T}$, that is, all subtraces involve the same protocol parameters. Lines 7 to 15 iterate through all calls in a subtrace. If a call returns an object that corresponds to a not yet bound protocol parameter, then this call binds the parameter (line 9). The $computeLabel()$ function computes a transition label for a call. The label contains the method signature, a protocol parameter denoting the receiver, and possibly also protocol parameters denoting method arguments and the return value. Arguments and return values are labeled if and only if they correspond to a protocol parameter, that is, if they appear as the receiver of a call in the subtrace.

After updating the list of recently called methods, the algorithm computes the destination state for the current call. The $nextState()$ function maintains a map from pairs of recently called methods and sets of bound objects to states. If no state for a given pair $(lastK, boundParams)$ exists, $nextState()$ creates a new state and adds it to the set $\mathcal{S}$ of states. After visiting all calls in a subtrace, the algorithm marks the current state as final. Once all subtraces have been analyzed, the algorithm returns an API usage protocol.

Our approach to generate protocols scales well with the size and number of subtraces. First, the protocol miner must visit each subtrace only once and requires a single linear pass through each subtrace. Second, the only data structure to keep in memory while analyzing a set of subtraces is the currently constructed FSM. Its size is independent of the size and number of subtraces because the maximum number of states is a function of the number of distinct API methods called in a set of subtraces, the number of protocol parameters, and the parameter k.

When applying Algorithm 5, we find small values of k to be the most promising. All results reported in this thesis are obtained with $k = 2$.

Getting back to our running example, consider the subtrace with core object `list` in Figure 4.8. The protocol miner groups this subtrace together with other subtraces that involve a collection and an iterator and infers a protocol, such as Figure 4.2.

4.4.4 Driving Protocol Mining with Passing Tests

Dynamic protocol mining is inherently limited by the execution traces given to the miner, which in turn is limited by the input used to execute the program. Most programs come with a finite set of inputs, for example, given as unit tests. Producing more input requires manual effort, which reduces the overall usefulness of protocol mining.

We address this limitation by combining protocol mining with test generation. The idea is to analyze an execution of a generated test in the same way as one would analyze executions driven by otherwise available input. A benefit of this approach is that generated tests are a large and diverse source of inputs. Random test generation provides fresh inputs for different random seeds, which can trigger behavior not yet triggered with other random seeds [31].

A potential problem of generating tests for protocol mining is that artificial input may trigger illegal API call sequences that can lead to incorrect protocols. We address this problem in two ways. First, we feed only execution traces of passing tests into the protocol miner, not execution traces of failing tests. While a passing test does not guarantee to use the program in the intended way, we found that in practice, most illegal inputs are filtered because they would lead to an exception. Second, the execution trace contains only API calls issued in the program, not calls issued in the generated tests (a generated test can call API methods, for example, to create an API object before passing it as a parameter). That is, each API call the miner learns from is part of the program and not a call that was randomly generated by the test generator.

4.5 Protocol Checking

Given protocols inferred by the protocol miner, our analysis checks the analyzed program for violations of these protocols with a dynamic protocol checker. Runtime checking guarantees soundness and completeness: All reported protocol violations provably occur in the execution, and all protocol violations in the execution are reported. Unfortunately, a dynamic checker can only reveal a problem if the problem occurs in the analyzed program execution, that is, if the problem is triggered by the given input. For example, checking the execution of a manually created unit test suite against a set of protocols is unlikely to reveal many protocol violations because the test suite exercises well-tested paths of the program.

We address this limitation by driving the program with generated tests. In contrast to the mining step of our approach, we now use generated tests that fail. These tests are classified as failing because they trigger a problem in the program that leads to an exception. Many failing tests do not expose any bug but fail because the generated test uses the program incorrectly. For example, a test may fail with a `NullPointerException` or an `IllegalArgumentException` because the test passes `null` or another illegal value as a method argument. Such tests are not relevant for programmers, as they do not reveal bugs in the program. To focus on tests that fail because of a bug in the program, we check whether the program violates a protocol while executing a failing test. If the program violates a protocol and if this viola-

```
Iterator someIterator = ...   // an iterator for some unrelated collection
Collection coll = new HashSet();
if (someIterator.hasNext()) someIterator.next();
```

Figure 4.9: Program to be checked against the protocol in Figure 4.2.

tion raises an exception, then the test exposes an unsafe API usage and therefore is relevant for programmers. In other words, we use protocol checking as a filter that identifies relevant warnings among all failing tests.

4.5.1 Naive Approach

An important question for checking programs against protocols is: What is a protocol violation? Since protocols are represented as FSMs, the naive answer is to check whether a sequence of API method calls is accepted by the FSM, that is, whether the sequence is a sentence in the language described by the FSM. However, this naive approach produces many more protocol violations than one might expect, in particular for multi-object protocols.

To understand why the naive approach fails, consider checking the program in Figure 4.9 against the protocol in Figure 4.2. The `someIterator` variable refers to an iterator over some collection not shown in the example. Instantiating the `HashSet` and assigning it to `coll` and brings the protocol from its initial state to state 2. The next call in the program is `hasNext()`, but state 2 does not have an outgoing transition labeled with `hasNext()`. Should the checker report a warning? Obviously, it should not report a warning because we have not yet created an iterator for `coll` and because `someIterator` is unrelated to `coll`.

4.5.2 Setup Phase versus Liable Phase

The problem illustrated by the example in Section 4.5.1 is that some states of inferred protocols do not fully specify which calls are legal. For example, state 2 does not specify whether it is legal to call `hasNext()`. We address this problem by distinguishing two phases of each protocol. At first, there is a *setup phase*, in which objects are bound to protocol parameters. Once all protocol parameters are assigned to an object, the protocol enters the *liable phase*. Only in the liable phase the checker should report warnings when the FSM of the protocol does not accept the observed method calls.

Distinguishing between a setup phase and a liable phase can be implemented in different ways. For the protocols produced by our protocol miner, the set of bound parameters in a particular state is unambiguous because the algorithm to create FSMs creates a fresh state whenever another protocol parameter is bound (Section 4.4.3). That is, we can partition the states of a protocol into *setup states*, where at least one protocol parameter is unbound, and *liable states*, where all parameters are assigned to an object. With this partitioning of states, checking whether a protocol instance is in the setup phase or in the liable phase reduces to knowing the current state.

Based on a partitioning of states into setup states and liable states, the checker avoids reporting false warnings reported by the naive checking approach. For the protocol in Figure 4.2, states 1 and 2 are setup states, and states 3, 4, and 5 are liable states. With this state partitioning, the checker does not report a warning for the `hasNext()` call in Figure 4.9 because the call occurs in state 2, which is a setup state.

4.5.3 Checking Approach

Conceptually, the checking approach is similar to existing runtime verification approaches, such as JavaMOP [27]. Our implementation uses the infrastructure built for mining protocols. At first, the execution trace is split into subtraces in the same manner as for protocol mining (Section 4.4.2). Then, each subtrace is verified against all protocols that have protocol parameters with types equal to the receiver types in the subtrace. The checker assigns each receiver object in the subtrace to a protocol parameter of matching type. For subtraces with multiple receiver objects of the same type, the checker creates multiple protocol instances, one for each possible assignment of objects to protocol parameters.

At the beginning of a subtrace, the protocol is in the initial state. The checker goes through all calls in the subtrace and analyzes all calls that appear in the alphabet of the protocol. Each such call is matched against the outgoing transitions of the current state. If a transition matches, the current state is set to the target state of the transition. If no transition matches and the current state is a liable state, the checker reports a protocol violation. Non-matching transitions in setup states are ignored.

For example, consider the inferred protocol in Figure 4.1 and the source code in Figure 4.4. Recall that class `C` uses a stack in an unsafe way because a client of `C` can trigger a violation of the stack protocol via `C`'s public interface. Our approach finds this problem because Randoop generates a test that triggers an API protocol violation by calling `C`'s methods. The test calls `report()` without a preceding call to `fill()` and therefore triggers a call to `peek()` while the stack is empty. This call causes an `EmptyStackException` and fails the test. Figure 4.10 shows the relevant subtrace of the test's execution trace. The subtrace contains all calls to the stack that were observed during the test execution. The checker finds a violation of the protocol in Figure 4.1 because the second call in the trace tries to call `peek()` at state 2, but `peek()` is only allowed at state 3.

```
... --> Stack()
... <-- Stack(): Stack{s}
... --> Stack{s}.peek()
# EmptyStackException
#   at Stack.peek()
#   at C.get()
#   at C.report()
#   ...
```

Figure 4.10: Subtrace from executing a test that triggers the unsafe API usage in Figure 4.4.

4.6 Warnings without False Positives

Reporting few (ideally no) false positives is crucial to make a bug finding technique applicable in practice [13]. By using runtime checking, our approach avoids a problem typically encountered with static checkers, namely false positives due to approximations of execution paths. Another potential source of false positives are inferred protocols that disallow legal behavior because this behavior was not observed during protocol mining. Our approach eliminates this problem by only reporting those protocol violations that certainly lead to an exception thrown by the API, that is, to undesired behavior. For example, the call to `peek()` in Figure 4.4 violates a protocol and results in an `EmptyStackException`, a clear indication that the API is used incorrectly.

To filter protocol violations that cause an exception, we need several pieces of information. For each failing test, the analysis logs the reason for the failure, which consists of two parts: the type T_{exc} of the thrown exception and the stack trace $S = (loc_1, loc_2, \ldots, loc_n)$ when throwing it. Each stack trace element loc_i indicates a source code location. For each protocol violation, the protocol checker reports the source code location loc_{viol} where the violating API call occurs. Furthermore, the analysis statically extracts the set $\mathcal{D}_{API}$ of declared exception types of the API method that is called at loc_{viol} and the set $\mathcal{D}_{program}$ of declared exception types of the method containing loc_{viol}. An exception can be declared using the `throws` keyword or using the `@throws` or `@exception` Javadoc tags.

Our analysis reports a protocol violation to the user only if the following three conditions are true:

1. $T_{exc} \in \mathcal{D}_{API}$
2. $loc_{viol} \in S$
3. $T_{exc} \notin \mathcal{D}_{program}$

The first two conditions ensure that the call that violates the protocol is responsible for failing the test. The third condition avoids warnings for methods that deliberately pass on exceptions thrown by the API to their own clients. In this case, the protocol-violating class is implemented safely because it explicitly passes on responsibility for using the API correctly to its clients. For Figure 4.4, all three conditions hold and our analysis reports an unsafe API usage, along with a test case that illustrates how to trigger the problem.

In principle, the approach might report a false warning when two conditions hold. First, an API method `m()` must be implemented incorrectly and throw an exception despite being called legally. Second, a mined protocol must forbid calling `m()` at a state where a permissive protocol [72] would allow it. One of the two conditions alone does not lead to false warnings. In particular, incomplete protocols alone, which may result from dynamic protocol mining, do not cause false positives. In practice, both conditions together are very unlikely to occur and we did not encounter this situation during our experiments.

In practice, the question whether a warning corresponds to a bug that should be fixed depends on many influences that go beyond what a program analysis can check. In this work, we assume that classes should use APIs safely, as described in Section 4.2.1. Based on this assumption, developers can fix a true positive reported by the analysis in two ways. First, the developers can modify the API usage in such a way that the API protocol is always followed, that is, the developers ensure that no exception of type T_{exc} is propagated to clients. Second, the developers can make explicit that an exception due to an API protocol violation may be propagated to clients, that is, the developers extend $D_{program}$.

4.7 API-Guided Test Generation

The preceding sections describe how randomly generated tests can drive a program for mining API protocols and for checking the program against inferred protocols. With a random test generation tool, such as Randoop, many generated tests do not trigger API calls. The reason is that Randoop chooses which method to call randomly from all methods in a program, even though only a subset of all methods uses an API. This section presents a heuristic to optimize our approach by guiding random test generation towards methods that trigger API calls.

Given infinite time, standard Randoop triggers the same behavior as a guided approach that focuses on methods relevant for calling the API. In practice, increasing the number and variety of API calls triggered by generated tests within a finite amount of time is important for two reasons. First, random test generators require a stopping criterion—typically, wall clock time or the number of generated method call sequences. Second, random test generators execute arbitrary code, which makes them susceptible to crashes that are hard to avoid in general [116]. Whatever stops the test generator, triggering API calls earlier than a purely random approach not only saves time, but also is likely to trigger more API usage bugs before the test generator terminates.

To guide Randoop towards methods relevant for triggering API calls, we statically analyze the program and prioritize its methods. The priorities influence the random decisions of the test generator in such a way that calling a method with higher priority is more likely. Methods that do not contribute at all to calling the API get a priority of zero and are ignored altogether by the test generator.

Computing Method Priorities

We build upon two well-known graph representations of the program under test: an inverted, context-insensitive call graph and a parameter graph [36]. The inverted call graph is a directed graph where vertices represent methods and edges represent a "may be called by" relationship between methods. We extract call graphs with Soot [157]. The parameter graph is a directed graph where vertices also represent methods and where edges represent a "can use the result of" relationship between methods. In the parameter graph, there is an edge from m_1 to m_2 if m_1 requires an object of type T (as receiver or as argument) and if m_2's return type is equal to T or a subtype of T.

Test generators call methods for three reasons. First, because a method is among the *methods under test*. In our case, these are all methods that call the API. Second, because the method returns an object that can be passed as an argument to another method, that is, the method is a *parameter provider*. Third, because the method may change the internal state of an object that is used for another call afterwards, that is, the method is a *state changer*. Our prioritization technique considers these three reasons and computes for each method three priority values that indicate how relevant

Algorithm 6 Compute priorities for calling methods during random test generation

```
Input: Inverted call graph $G_c$, parameter graph $G_p$, methods $\mathcal{M}$, API methods $\mathcal{M}_{API}$
Output: Map $p : method \to \mathbb{R}$ assigning priorities to methods
 1: initialize $p_{mut} : method \to \mathbb{R}$ to zero        ▷ priorities for methods under test
 2: for all $m \in \mathcal{M}$ do
 3:     for all $m_{API} \in \mathcal{M}_{API}$ with $d_c(m_{API}, m) \le d_c^{max}$ do
 4:         $p_{mut}(m) \leftarrow p_{mut}(m) + \frac{relevance(m_{API})}{d_c(m_{API}, m)}$
 5: initialize $p_{param} : method \to \mathbb{R}$ to zero        ▷ priorities for parameter providers
 6: for all $m \in \mathcal{M}$ with $p_{mut}(m) > 0$ do
 7:     for all $m' \in reachable(G_p, m)$ do
 8:         $p_{param}(m') \leftarrow p_{param}(m') + \frac{p_{mut}(m)}{nbProviders(retType(m'))}$
 9: initialize $p_{state} : method \to \mathbb{R}$ to zero        ▷ priorities for state changers
10: for all $m \in \mathcal{M}$ with $p_{mut}(m) > 0$ do
11:     for all $t \in inputTypes(m)$ do
12:         for all $m' \in methods(t)$ do
13:             $p_{state} \leftarrow p_{state} + \frac{p_{mut}(m)}{|methods(t)|}$
14: $p \leftarrow merge(p_{mut}, p_{param}, p_{state})$
```

the method is for each reason. Afterwards, the priorities are combined into a single priority value per method.

Algorithm 6 describes how we compute the priority of each method. The algorithm takes as input the call graph G_c, the parameter graph G_p, the set $\mathcal{M}$ of all methods, and the set $\mathcal{M}_{API}$ of API methods. There are four main steps.

Priorities for Methods Under Test At first, we compute how relevant a method is depending on whether it calls any of the API methods. Methods calling an API method directly are the most relevant ones. In addition, methods calling API methods indirectly, that is, via other methods, are given a lower priority. The reason for considering indirect calls is that some methods should be called at a state built up by another method. For instance, a method m_1 may create a file and call another method m_2 that writes into the file. By calling m_1, the test generator can trigger successful writes into the file, even though m_1 calls the file writing API only indirectly. We limit the level of indirection up to which to consider indirect callers of API methods to d_c^{max}, which is set to three in our experiments. The priority gain of a method calling an API method depends on two factors. First, it increases with the relevance of the called API method:

$$relevance(m_{API}) = \frac{1}{|\{m \mid (m_{API}, m) \in G_c\}|}$$

Less frequently called methods are more relevant so that all API methods get an equal chance of being triggered. Second, the priority gain decreases with the minimal distance $d_c(m_{API}, m)$ in G_c between the API method m_{API} and the calling method m. The rationale for this decrease is that the call graph contains may-call and not must-call edges; fewer indirections to an API call increase the chance that the API is actually called during an execution.

Priorities for Parameter Providers We compute how relevant a method is for providing parameters for calling methods under test. Therefore, we consider all methods $reachable(G_p, m)$ that are reachable from a method under test m in the parameter graph. These are all methods that directly or indirectly contribute parameters for calling the method under test. The gain of priority for such a parameter provider depends in two factors. First, it increases with the priority of the method it provides parameters for. Second, the gain of priority decreases with the overall number of providers for the type required by the method under test. The rationale for this decrease is to give higher priority to methods providing an uncommon type than to methods providing a type returned by many other methods.

Priorities for State Changers The third kind of priority is for methods that change the internal state of objects used for calling methods under test. The types $input\text{-}Types(m)$ required for calling a method m are the receiver type and all argument types of m. We consider each method that is provided by an input type for a method under test as a potentially state-changing method and therefore increase the method's priority. The priority gain depends on two factors. First, it increases with the priority of the method under test that the state changer may influence. Second, the priority gain decreases with the overall number of methods provided by the input type. The rationale for this decrease is to give objects of each input type an equal chance of being changed, independently of the number of methods of the type.

Merging The final step is to normalize the three kinds of priorities and to combine the three priorities of each method in a weighted sum. The weights indicate how important calling methods under test, calling parameter providers, and calling state changers are, respectively. In our experiments, the test generator devotes 20% of all calls to calling state changers and 40% to each calling methods under test and to calling parameter providers. These weights are selected based on initial experiments and may not be optimal.

Example

Figure 4.11 shows an example to illustrate how prioritizing methods guides random test generation towards calling API methods. The example contains seven methods and constructors, of which one, `doIt()`, calls the API. Algorithm 6 finds this method as the only method under test and assigns a high priority to it. To call `doIt()`, the test generator requires an object of type `A`, which in turn requires objects of types `B` and `C`. Our algorithm finds the constructors of these classes as parameter providers and gives them a medium priority. The API call in `doIt()` results in a NullPointer-Exception unless `init()` is called beforehand. This method is found to be a state changer and also gets a non-zero priority. The remaining two methods, `B.m()` and `D()`, do not contribute to calling the API and get a priority of zero. Given the priorities, the random test generator selects only methods that eventually contribute to

```
class A {
  API api;
  A(B b) { .. }
  void init() { api = APIPool.some; }
  void doIt() { api.use(); }
}
class B {
  B(C c) { .. }
  void m(D d) { .. }
}
class C {}
class D {}
```

Method	Priority
`A.doIt()`	107
`A()`	33
`B(C)`	27
`C()`	27
`A.init()`	7
`B.m(D)`	0
`D()`	0

Figure 4.11: Example for prioritizing methods towards using the API.

calling the API and probabilistically leads to more calls to the API than a purely random approach. We show this claim to be valid for real-world programs and APIs in Section 4.8.3.

4.8 Evaluation

Our evaluation was driven by two main questions:

1. *How effective is our approach in finding unsafe API usages?* We find 54 unsafe API usages in ten well-tested Java programs. Manual inspection shows all of them to be true positives.
2. *How much does the testing effort reduce compared to a purely random approach by guiding test generation towards an API?* Our heuristic optimization reveals bugs with five times fewer generated call sequences than the unguided approach, on average. This improvement is possible because method prioritization increases the number of API calls by a factor of 57 and decreases the sequences required to call a particular API method by a factor of seven.

4.8.1 Setup

We run our analysis on all programs from the DaCapo benchmarks, version 2006-10-MR2 (Table A.1) [14]. The analysis focuses on API classes of the Java library (Table 4.1).

We run the analysis in two modes: *unguided* and *guided* towards an API. Unguided runs use Randoop's standard test generation technique. Guided runs use method prioritization (Section 4.7). To evaluate guided runs, we run Randoop for each program-API pair. To evaluate unguided runs, we run Randoop on each program. Since Randoop is based on random decisions, we perform each experiment with ten different random seeds [117]. Unless otherwise mentioned, the reported results are average values of all ten runs.

Randoop requires a stopping criterion. We use a maximum number of generated method call sequences of 10,000. We prefer this criterion over a maximum running time because it is easily reproducible, whereas Randoop's execution time depends on the environment used for the experiments.

4.8.2 Detection of Protocol Violations

The analysis finds a total of 54 protocol violations. Each static source code location with a protocol violation counts only once, even if it occurs in multiple failing tests. Table 4.2 shows the number of violations detected for each program-API pair, along with the total number of violations found over all runs. The table also shows the number of failing tests among which the protocol checker finds the bugs. On average,

Table 4.1: Analyzed APIs.

API	Classes
Coll/Iter	All public methods of `java.util.Collection`, `java.util.Iterator`, and their subclasses.
Vector/Enum	All public methods of `java.util.Vector`, `java.util.Enumeration`, and their subclasses.

Table 4.2: Number of failing tests (out of 10,000) and number of detected API usage bugs.

Program	Failing tests (Average per run)		Bugs found		
			Average per run		Total
	Coll/Iter	Vector/Enum	Coll/Iter	Vector/Enum	
ANTLR	1,429	1,695	0	0	0
BLOAT	4,098	4,042	0	0	0
chart	2,336	2,396	1.4	1	4
Eclipse	2,817	3,375	0	0	0
FOP	3,148	2,948	3.2	2.8	9
HSQLDB	2,246	2,264	0	0.1	1
Jython	4,314	5,169	1.4	0	2
Lucene	3,573	3,372	1.9	1.1	5
PMD	2,155	4,158	12.8	5.9	16
Xalan	4,440	4,395	6.6	5.9	17
Sum	3,056	3,381	27.3	16.8	54

3,219 of 10,000 tests fail—too much for a programmer to inspect. The protocol checker filters these tests and presents only those that expose a protocol bug.

We inspect all protocol violations manually to verify that no false positives are reported. Indeed, all reported violations show an illegal API usage that can be triggered via the public methods of a class in the program. Furthermore, we manually check whether the comments of the buggy methods contain any natural language warnings about the exceptions that may result from these methods. None of the reported methods has such a comment. That is, all reported bugs are real problems, correctness and documentation issues, in the programs. Table B.2 lists details on all detected protocol violations.

Figure 4.12 shows a bug found in FOP. The class `MIFDocument` contains an inner class `BookComponent` that stores pages in a list, which initially is empty. The inner class provides a method `curPage()` that tries to return the last page in the list. This method is called in the public method `setTextRectProp()`, which raises an `IndexOutOfBoundsException` when the list of pages is empty. This risk is not apparent to clients of `MIFDocument`, and hence, the class uses `ArrayList` in an unsafe way. Our analysis finds this bug because the mined protocol for `ArrayList` disallows `get()` directly after creating the list.

Figure 4.13 is a bug found in Jython. The class has an `Iterator` field that is accessed by multiple public methods. One of them, `_flushCurrent()`, calls `remove()` on the iterator. The iterator protocol allows this call exactly once after each call to `next()`. This constraint is not respected in the class, causing a potential `IllegalStateException` when calling `_flushCurrent()`. Similar to the first example, the class uses an API in an unsafe way and does not make the potential problem apparent to its clients. Our analysis infers the iterator protocol and finds this violation of it.

For the examples in Figures 4.12 and 4.13, one may argue that the API-using classes `MIFDocument` and `InternalTables2` have protocols on their own. Un-

```java
public class MIFDocument {
    protected BookComponent bookComponent;
    public MIFDocument() {
        bookComponent = new BookComponent();
    }
    public void setTextRectProp(int left, int top,
                                int width, int height) {
        (bookComponent.curPage()).curTextRect()
           .setTextRectProp(left, top, width, height);
    }
    public void createPage() {
        bookComponent.pages.add(new Page());
    }
    class BookComponent {
        ArrayList pages = new ArrayList();
        private Page curPage() {
            return (Page)pages.get(pages.size() - 1);
        }
    }
}
```

Figure 4.12: Incorrect usage of `ArrayList` in FOP.

fortunately, these classes do not make their protocols explicit to clients of the classes. Instead, these classes propagate API protocol violations to clients, which may not even know about the API. The safety property that our analysis checks (safe API usage, see Section 4.2.1) ensures that if API developers deem a protocol violation important enough to declare a potential exception, then API-using classes should also do it.

An obvious question about bug reports is how to fix the bugs. During our experiments, we have seen two strategies towards safe API usage. First, an API-using method `m()` can ensure to follow the protocols of all API objects that `m()` uses and not to propagate any exception from the API to callers of `m()`. Often, such methods return a special value, for example `null`, when they cannot return another value. Second, an API-using method `m()` can propagate API exceptions and declare these exceptions in its signature. This approach indicates to callers of `m()` that calling the method at an illegal state or with illegal input may result in an exception, and hence, `m()` explicitly passes the responsibility on to its callers.

Interestingly, many of the detected bugs are surrounded by attempts to shield users of a class from protocol violations. For example in Figure 4.13, the `_next()` method checks by calling `hasNext()` whether a call to `next()` is legal and returns `null` otherwise. That is, the programmers protect clients of the class from some violations of the iterator protocol but, unfortunately, forgot about another violation.

Comparison to Existing Approaches

We directly compare our analysis to OCD [55], a combined dynamic protocol miner and checker. OCD also searches for protocol violations by mining and checking pro-

```
public class InternalTables2 {
    protected Iterator iter;
    public void _beginCanonical() {
        iter = ((TableProvid2)classes).values().iterator();
    }
    public Object _next() {
        if (iter.hasNext()) {
            cur = iter.next();
            return (PyJavaClass)cur;
        }
        return null;
    }
    public void _flushCurrent() {
       iter.remove();
       classesDec(((PyJavaClass)cur).__name__);
    }
}
```

Figure 4.13: Incorrect usage of `Iterator` in Jython.

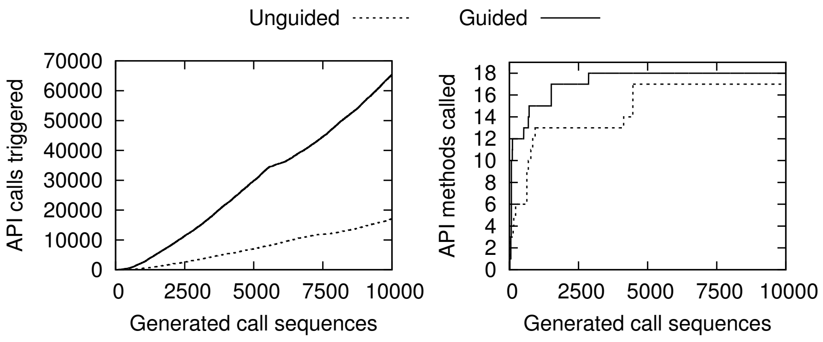
tocols at runtime. Two main differences are that OCD relies on existing input to drive the program and that the analysis combines mining and checking into a single run of the program under test. Gabel and Su used OCD to analyze the usage of the Java standard library in the DaCapo 2006-10-MR2 programs, which we also analyze here. Their technique reports three warnings, matching Gabel and Su's expectation to find few problems in these well-tested benchmarks. Manual inspection showed two warnings to be clear false positives and one warning to be a potential problem.

Applied to the same programs, our analysis reports 54 warnings that are all true positives. What are the reasons for these differences? First, our analysis does not report false positives by construction, as explained in Section 4.6. Second, OCD is limited to the program paths triggered by available input (here, DaCapo's benchmark input). In contrast, the generated tests used by our analysis exercise less intensively tested parts of the programs and trigger API usage bugs not exposed with the benchmark inputs.

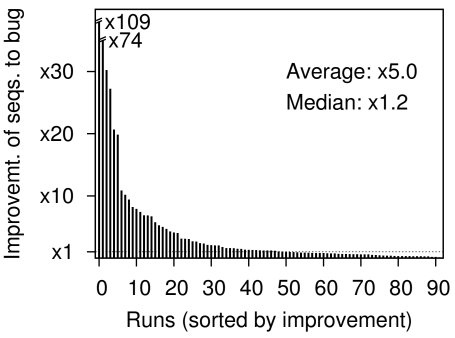
We also compare our approach to FindBugs [75] by running it on the DaCapo 2006-10-MR2 programs. Although FindBugs reports various problems, it misses all bugs found by our analysis. The reason is that the list of bug patterns coming with FindBugs covers only a small subset of all possible API usage bugs, excluding those detected in our evaluation. Instead of relying on a pre-defined list of bug patterns, our approach extracts API protocols automatically, making it easy to apply the approach to arbitrary APIs.

4.8.3 API-Guided Test Generation

The goal of guiding test generation towards an API is to reduce the overall time required to reveal API usage bugs. The most important factor for reducing this time is the number of generated call sequences, which controls the running times of the



(a) Unguided vs. guided test generation for Xalan and Vector/Enum.



(b) Improvement (that is, reduction) of the number of call sequences required to trigger a bug for guided runs compared to unguided runs.

Figure 4.14: Graphical comparison of guided and unguided test generation.

test generator, the protocol miner, and the protocol checker. Therefore, we use the number of generated call sequences as a metric for testing effort.

We compare guided runs of our analysis to unguided runs (Table 4.3). Do guided runs trigger more API calls than unguided runs within a fixed number of generated call sequences? The first block of columns in Table 4.3 compares the number of API calls without guidance and with guidance. Each value is the average over ten runs of Randoop with different random seeds. For all but one program-API pair, guided runs trigger more API calls. For example, the number of calls to Vector/Enum triggered in tests for ANTLR increases from 6,146 to 58,438. On average over all programs, APIs, and runs, guidance improves the number of API calls by 56.7x (median: 5.2x).

Increasing the number of API calls is important to give the miner different examples to learn from and to increase the chance to hit a bug. In addition to increasing the number of API calls, it is also important to trigger calls to *different* API methods to expose a broad variety of API protocols and their violations. The second block of columns in Table 4.3 lists the number of distinct API methods called. In most cases, there is no significant difference between guided and unguided runs. The reason is that our stopping criterion lets the test generator run long enough to give even unguided runs a realistic chance to hit each API method.

How long does it take the test generator to hit an API method? The third block of columns in Table 4.3 shows after how many generated sequences an API method is called the first time. For all but one program-API pair, API methods are called earlier in guided runs than in unguided runs. On average over all programs, APIs, and runs, the guided approach calls a method 6.9 times faster than the unguided approach. That is, even though after enough time the same set of API methods is called, guided runs achieve this goal much faster than unguided runs.

Figure 4.14a illustrates an unguided and a guided run. The graphs show the cumulative number of API calls and the number of distinct called API methods depending on how many sequences Randoop has generated, respectively. The first graph shows that the number of API calls increases much faster with guidance than without guidance. The second graph illustrates that the guided run triggers API methods much faster than the unguided run, even though both arrive roughly at the same number of API methods after about 4,500 sequences.

Finally, does guided test generation trigger bugs earlier than unguided test generation? Figure 4.14b shows how the number of generated sequences required to trigger a bug improves (that is, reduces) through guidance. The figure compares all runs where a bug was found both with and without guidance. In most of the runs, bugs are triggered faster with guidance, with up to 109x improvement. For some runs, our heuristic triggers bugs slower, with a slowdown up to 8x. On average, guided runs improve upon unguided runs by 5x.

There is one outlier in Table 4.3: For Jython and the Coll/Iter API, guidance decreases the number of API calls and increases the sequences needed to trigger an API method. The reason is that almost all methods (91%) in Jython are relevant for using the Coll/Iter API, which skews the method priorities. After all, our guidance technique is a heuristic that, even though successful in most cases, cannot guarantee improvements for all programs and APIs.

4.8.4 Scalability and Performance

The total execution time of our analysis is the sum of three components: the time for generating and executing tests, the time for mining protocols from execution traces, and the time for checking execution traces against protocols. All three components scale well to large programs. Randoop's execution time is the product of the number of sequences to generate and a program-specific factor (because Randoop executes code from the program). The performance of the protocol miner scales linearly with the size of the execution traces [125], which is also true for the checker because it builds upon the same infrastructure. On a standard PC, analyzing a program-API pair takes between less than a minute and several minutes.

4.8.5 Threats to Validity

We are aware of the following threats to the validity of this evaluation. First, the selection of programs we analyze may not be representative for all Java programs. To avoid selection bias, we analyze all programs from a well-known benchmark suite [15] and report the results for all of them. Second, we stop the test generator at a maximum number of generated tests. A different stopping criterion may give different results. Third, the number of repetitions we use to compensate for the random nature of the test generator may not be sufficient to accurately characterize the real distribution of the underlying random process. Finally, the assessment that all reported warnings are true positives depends on the assumption that developers require safe API usages. If this safety property is not desired by the developers, for example, because the project is small enough to support undocumented propagation of exceptions, then the reported warnings may not be relevant.

Table 4.3: Comparison of unguided and guided test generation. The last row summarizes the results from all runs.

Program	API calls				Called API methods				Sequences to first call			
	Coll/Iter		Vector/Enum		Coll/Iter		Vector/Enum		Coll/Iter		Vector/Enum	
	Ung..	Guid.	Ung..	Guid.	Ung..	Guid.	Ung..	Guid.	Ung..	Guid.	Ung..	Guid.
ANTLR	3,068	7,915	6,146	58,438	3	3	11	11	930	510	651	188
BLOAT	33,363	47,269	0	0	62	62	0	0	686	470	0	0
chart	1,564,161	2,356,751	278	19,761	77	75	5	6	624	236	5,472	720
Eclipse	33,334	164,051	29,285	152,103	45	40	20	20	506	348	804	512
FOP	12,291	59,220	1,163	22,078	46	46	12	12	1,016	601	1400	260
HSQLDB	1,549	50,730	37,479	656,172	12	28	12	14	2,503	275	1,238	105
Jython	261,276	236,718	36,474	122,400	105	89	18	18	444	802	1,414	575
Lucene	70,884	136,490	66,169	97,585	38	37	12	12	561	537	942	1,861
PMD	16,371	134,074	2,934	450,228	91	93	19	21	1,264	638	6,051	2,167
Xalan	22,289	116,225	38,230	106,600	9	9	17	18	940	459	802	376
Avg./Med.	**Improvement: x56.7 / x5.2**				**Improvement: x1.0 / x1.0**				**Reduction: x6.9 / x1.9**			

4.9 Support for the Thesis

The presented approach for finding protocol bugs supports our thesis that automatic program analysis allows for precisely detecting programming errors with little effort.

Automation Our approach achieves a high degree of automation for two reasons:

- *Test generation*. Dynamic analysis is bound to analyze executions driven by available input. Traditionally, such input must be written by developers, for example, unit tests, or must be provided by users, for example, through record and replay of executions in the field. Instead of relying on human-produced input, we leverage test generation as a driver for dynamic analyses. A key insight of our approach is to distinguish passing from failing tests, to use the former to learn about common behavior, and to use the latter to check for violations of common behavior.
- *Protocol mining*. Checking a program against a specification requires a specification, which—traditionally—must be written by developers in addition to a program. Instead, our approach leverages protocol mining to obtain specifications automatically.

Precision Our approach is precise because all warnings that it reports are guaranteed to point to unsafe API usages, that is, true positives. Along with each warning, the analysis reports a concrete test case that shows how using a class leads to a protocol violation that surprises clients of the class. The reason why the approach avoids false positives is that it focuses on exceptions, that is, on certainly undesired behavior.

Effort The approach involves low human and computational effort. Human effort for using the approach is the sum of the effort to invest for starting the analysis and the effort to invest for inspecting its warnings. The effort for starting the analysis is small because of its high degree of automation. The only input required by the analysis are the classes of the program under test. The effort for inspecting reported warnings is small because all reported warnings are true positives and because each warning comes with a concrete test case.

The computational effort of the approach (Section 4.8.4) is low enough for an automatic program analysis that runs without any human interaction. For large-scale deployment, parallelizing the analysis is straightforward since each class-API pair can be analyzed independently from the others.

4.10 Limitations and Future Work

We envision several directions of future work:

- *Other kinds of specifications*. The current analysis is limited to detecting violations of API protocols. Future work may apply the approach to other kinds of dynamically inferred specifications, such as invariants [46, 68].
- *Non-exceptional misbehavior*. The current analysis focuses on bugs that manifest through an exception. While this approach guarantees all reported warnings to correspond to severe bugs, we might miss more subtle problems that lead to

incorrect but not obviously wrong behavior. Future work may adapt our approach to search for misbehavior that does not manifest through an exception.

- *Application-level feasibility of paths*. The current analysis reveals unsafe API usages in classes of a program but does not guarantee that a crash is feasible when using the program as a whole, for example, via its graphical user interface. Nevertheless, all reported bugs are real problems because a programmer can trigger them via the public interface of a class. That is, even if a bug is infeasible in the current program, changes in using the buggy class can make it feasible. Programmers can avoid this risk through safe API usage. Future work may use GUI-level test generation [64] to analyze paths that are feasible when using the program as whole.
- *Unfinished API usages*. The current analysis checks for method calls that are illegal at the current protocol state. In addition, a checker may search for API usages that do not reach a final protocol state, that is, unfinished API usages. A challenge for such an approach is to avoid false positives caused by generated tests that trigger incomplete usages not possible when using the program as a whole.

5 Incorrectly Ordered, Equally Typed Arguments

In statically typed programming languages, method parameters have types to ensure that each argument passed to a method has the expected type.[1] Unfortunately, type specifications are insufficient when a method has multiple parameters of the same type. For example, a method `setEndPoints(int high, int low)` requires two `int` arguments. If a programmer accidentally calls this method with incorrectly ordered arguments, the compiler has no means to warn her. Can we support programmers in ordering equally typed arguments correctly?

In this chapter, we present an automatic, mostly language-agnostic, static analysis to detect anomalies in the order of equally typed method arguments. The analysis takes the source code of a program as its input and reports warnings about potentially erroneous call sites of methods with equally typed arguments. In contrast to Chapters 2, 3, and 4, the analysis reports programming errors that go beyond correctness bugs. The analysis also reports warnings that point developers to naming bugs and other noteworthy anomalies. Warnings that do not directly affect correctness can still be relevant for developers because these warnings point to problems related to the understandability and maintainability of a program.

In the following section, we motivate the analysis with real-world programming errors related to equally typed arguments. Sections 5.2 to 5.4 present the analysis and its implementation for Java and C. In Section 5.5, we evaluate the approach by applying it to real-world programs. Finally, Sections 5.6 and 5.7 discuss how the analysis fits the thesis of this work and outlines directions of future work.

5.1 Motivation

There are three kinds of programming errors related to equally typed method arguments, which we illustrate with examples from real-world Java and C programs.

- A programmer can accidentally reverse arguments and pass them in the wrong order (Figure 5.1a). Such a mistake leads to unexpected program behavior and

[1] We refer to formal parameters in a method declaration as parameters and to objects passed to methods at a call site as arguments. When saying method, we mean both functions (as in C) and methods (as in Java).

affects the program's *correctness*. To correct this kind of error, the programmer should reorder the arguments.

- Equally typed method parameters with badly chosen parameter names make using a method unnecessary difficult (Figure 5.1b). Identifier names play an important role for program understanding [89] and code quality [24]. Since this is particularly true for equally typed method parameters, inadequate names affect the program's *understandability*. To correct this kind of error, the programmer should give the parameters more meaningful names.
- Arguments that are unusually ordered can confuse a reader of the source code. An unusual argument order can be necessary, for example, because the program's semantics require doing the inverse of the expected (Figure 5.1c). However, an unusual argument order naturally raises the question whether a method call site is correct. Unless a comment explains the reason for such an anomaly, it will negatively affect the program's *maintainability*. To correct this kind of error, the programmer should add a comment that explains why an unusual argument order is correct.

Programming errors related to equally typed arguments are hard to find. The main reason is that these problems involve the semantics of the program, which are not explicit in the source code but only exist in the mind of the programmer. Traditional compilers are oblivious to the order of equally typed arguments: As long as the types of arguments and parameters match, the program compiles without warnings. The problem is compounded by the fact that bugs caused by incorrectly ordered arguments may not raise an exception, and therefore remain unnoticed during testing. For instance, reversing the arguments of a call to `setEndPoints(int high, int low)` introduces a subtle semantic error, which can remain unnoticed until late in the development process.

Call sites of methods with equally typed arguments account for a significant part of all method call sites. Within a corpus of programs comprising 1.6 million lines of Java code (DaCapo 9.12, Table A.2), 11% of all method call sites (77,610 out of 683,504) have two or more equally typed arguments. That is, for 77,610 method call sites the type system cannot ensure that the arguments passed by the programmer are ordered correctly. The problem is even more relevant for C programs. In a collection of 620 thousand lines of C code (Spec CPU 2006, Table A.3), 26% of all call sites (25,219 out of 96,633) have equally typed arguments. On average, these programs contain a call site with unchecked argument order every 24 lines of code. As evidenced by various entries in public issue tracking systems and source code repositories[2], programmers are susceptible to problems related to equally typed arguments.

To the best of our knowledge, there is no existing technique to automatically find anomalies related to equally typed arguments. However, there exist two kinds of approaches to prevent argument ordering problems. The first approach are conventions. For example, the arguments of a method moving data from a source to a sink are typically ordered so that the source argument is passed before the sink argument. Conventions can prevent argument ordering bugs, but require careful and

[2]For example, see issue 4732 in the Apache Hadoop Common bug tracker, issue 3890 in the Liferay bug tracker, revisions 58536, 58764, and 60357 of the JBoss SVN repository, or revisions 10263 and 13935 of the JikesRVM SVN repository.

disciplined programming. Also, there are cases where no obvious ordering of arguments exists, and hence, conventions are hard to apply in practice. The second approach to prevent argument ordering problems is better support by the programming language. Some languages, such as Smalltalk and Scala, allow for named arguments, where callers of a method explicitly assign arguments to method parameters. For example, one can call `setEndPoints(high=myHigh, low=myLow)`. However, named arguments are not available in all languages, and also introduce additional boilerplate code, which may not be accepted by programmers.

	(a)	(b)	(c)
Program	Eclipse 3.5.1	Jython 2.5.1	gcc 3.2

Method call	`createAlignment(name, mode,`
	`Alignment.R_INNERMOST,`
	`count, sourceRestart,`
	`adjust);`
		`_pow(coerce(left), value, null)`	`generateOptimizedBoolean(`
			`currentScope, codeStream,`
			`falseLabel, trueLabel,`
			`valueRequired)`

Called method	`Alignment createAlignment(`
	`String name, int mode,`
	`int count, int sourceRestart,`
	`int continuationIndent,`
	`boolean adjust)`
		`PyFloat _pow(`
		`double value, double iw,`
		`PyObject modulo)`
			`void generateOptimizedBoolean(`
			`BlockScope currentScope,`
			`CodeStream codeStream,`
			`Label trueLabel, Label falseLabel,`
			`boolean valueRequired)`

Comment	Bug caused by incorrect argument ordering: The highlighted arguments are not at the expected position. Triggered by our bug report, the problem has been fixed for Eclipse 3.7.	Badly chosen parameter names: The method performs exponentiation of two `double` parameters. Renaming the first two parameters to `base` and `exponent` would clarify their semantics.	Noteworthy anomaly: `if_true_label` and `if_false_label` are passed in the inverse order of the method declaration. A comment explaining this anomaly makes maintaining the code easier.

Figure 5.1: Examples of problems related to equally typed method arguments found in Java and C programs.

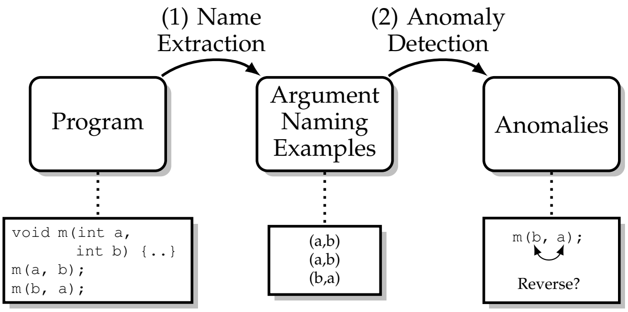
5.2 Overview of the Approach

We present a static program analysis to detect anomalies in the order of equally typed method arguments. The presented analysis is fully automatic and requires no input except for the source code of the program to analyze. Instead of relying on additional information, such as formal specifications, our technique infers knowledge about equally typed arguments from the source code. The output of the analysis is precise: Most of the reported anomalies are indeed programming errors. In experiments with well-tested Java and C programs, the analysis reports 66 warnings, out of which 54 (82%) point to problems that developers should address.

The key observation that enables our approach is that programmer-given identifier names convey implicit semantic knowledge about arguments. Our analysis leverages this knowledge by searching for inconsistencies in the names given to method arguments and method parameters. The analysis extracts identifier names from the source code of a program and compares the names used at different call sites of a method with each other using string similarity metrics. If reordering equally typed arguments at a particular call site fits the names used at other call sites of this method significantly better, our system reports an anomaly and proposes to reorder the arguments.

The analysis detects the three kinds of programming errors described in Section 5.1. The analysis finds correctness bugs caused by accidentally reversed arguments, such as Figure 5.1a, because the names of these arguments often deviate from the names of correctly ordered arguments. The analysis also reveals badly chosen parameter names (or naming bugs), such as Figure 5.1b, as these names often do not allow for inferring the correct argument order. Finally, the analysis detects noteworthy anomalies, such as Figure 5.1c, where reordering the arguments seems more in line with other call sites of the method than the current argument order. The analysis currently cannot distinguish whether a noteworthy anomaly is already commented or not. If there is no comment to explain the anomaly, the warning points to a programming error because the developers should add a comment. Suppressing warnings for noteworthy anomalies that have already a comment is left as future work.

Our approach consists of two steps (Figure 5.2). The first step, *name extraction*, gathers identifier names that programmers have given to method arguments and



```
void m(int a,
       int b) {..}
m(a, b);
m(b, a);
```

```
m(b, a);
```

Figure 5.2: Overview and a simple example.

method parameters. The output of this step is a list of *argument naming examples* for each method with equally typed arguments. These naming examples provide insights into the semantics of arguments and can be used to determine the order in which arguments should be passed. The second step of the analysis is *anomaly detection*. It searches for anomalies in the naming examples by computing the similarities between names used at different positions. An *anomaly* occurs if the names of arguments deviate from typically used names and if a different argument order than the order in the source code seems appropriate. The output of the second step is a list of anomalies, each coming with a proposal how to reorder arguments to avoid the anomaly.

The two steps of the analysis can be viewed as a front end and a back end. While the front end, which extracts naming examples from source code, is language-dependent, the back end, which searches for anomalies, is language-independent. A benefit of this separation is that one can easily adapt our approach to different programming languages. We present front ends for Java and for C.

We envision two usage scenarios for our approach. During the development of a program, the analysis provides an inexpensive, automatic technique to find problems related to equally typed arguments in an early stage of development. For example, if a programmer accidentally reverses two arguments, our analysis can spot this anomaly and report a warning even before testing the source code. Another usage scenario is maintenance of mature and well-tested programs. While in this scenario, we expect few bugs to be found, anomalies are nevertheless of interest, for example, to add a comment explaining why an unusual order of arguments is correct in a particular context.

5.3 Name Extraction

The goal of the name extraction step of our analysis is to gather as many examples as possible that show how programmers name the arguments passed to a method. We extract these examples from source code by analyzing method call sites and method declarations. As this work focuses on problems related to equally typed arguments, only methods with multiple parameters of the same type are considered.

The analysis traverses the abstract syntax tree and extracts from each method call site two kinds of information: the signature of the called method and the names of the arguments passed to the method. The part of the analysis that extracts names from arguments is language-specific. In each language, different kinds of expressions can be passed as arguments. In the following, we present name extraction techniques for Java and C. We begin with kinds of argument expressions supported by both Java and C, and afterwards discuss more specific argument expressions.

5.3.1 Java and C Language

The analysis extracts names from the following expressions:

- Identifiers (for example, local variables): The name of an identifier is simply the identifier itself.

- Array accesses: The name of an array access is the name of the array expression, that is, ignoring the index expression.

- Casts: The name of a cast expression is the name of the casted expression, ignoring the type to which it is cast.

5.3.2 Java Language

Our analysis extracts names from the following Java-specific argument expressions:

- Field accesses: The name of a field access is the name of the accessed field, ignoring the underlying expression on which the field is accessed. This includes fields of `this` and super fields.
- Method call sites (with return value passed as an argument): The name of a method call site is the name of the called method, ignoring the underlying expression that yields the method receiver. In Java, getter methods are a common naming practice. As the `get` prefix does not convey any semantics relevant for our approach, we remove this prefix from all method names starting with `get`.

For instance, the following Java method call sites provide three naming examples:

```
setEndPoints(x.highEP[i], lowEP);
// (highEP,lowEP)
setEndPoints(obj.h, getLow());          // (h,Low)
setEndPoints(getHighs()[5], (int) low) // (Highs,low)
```

5.3.3 C Language

For C programs, the analysis extracts the following names in the addition to those described in Section 5.3.1:

- Access to members of structs and unions: The name of a member access is the name of the accessed member, ignoring the underlying expression on which the member is accessed.
- Address-of operators: The name of an expression of the form `&x` is the name of `x`.
- Dereference operators: The name of an expression of the form `*x` is the name of `x`.
- Method call sites (with return value passed as an argument): The name of a method call site is the name of the called method, ignoring the underlying expression that yields the receiver.

For example, consider the following C method call sites and the extracted naming examples:

```
setEndPoints(h, low[3]);           // (h,low)
setEndPoints(foo->high, &low);     // (high,low)
setEndPoints(*highPtr, f().low()); // (highPtr,low)
```

Some argument expressions, such as literals and arithmetic expressions, are not analyzed because the analysis cannot extract unambiguous names from them. If at least one argument of a method call has no unambiguous identifier name, the analysis ignores the method call.

Besides call sites of methods, there is another source of information about the names of method arguments: formal parameters. Formal parameter names given in the declaration of a method are often similar to the names used at call sites. Therefore, we analyze all method declarations in a program and use formal parameter names as an additional example of how arguments are named. For example, the following method declaration gives a naming example:

```
void setEndPoints(int high, int low) {..} // (high, low)
```

The analysis groups naming examples so that all examples for the same method signature and for the same argument type are in one group. Grouping by method signature is useful because the argument names of one method are independent of the argument names of other methods. Overloaded methods are treated as different methods because one cannot easily map their parameters to each other. For instance, the following two variants of `m()` are treated as two methods, as we do not know how to map `a` and `b` to `x`, `y`, and `z`:

```
void m(int a, int b) {..}
void m(int x, int y, int z) {..}
```

Grouping by argument type is required because some methods expect equally typed parameters of multiple types. For instance, the following method expects two `int` parameters and two `String` parameters:

```
void m(int length, int offset, String name, String msg) {..}
```

In this case, we analyze naming examples for `m()`'s `int` arguments separately from naming examples for `m()`'s `String` arguments.

In summary, the naming examples extracted by the first step of our analysis are defined as follows:

Definition 12 (Argument naming examples)**.** *The* argument naming examples *of a method* `m()` *and a type* `T` *consist of the set* $\{N_{c_1}, .., N_{c_k}, N_{decl}\}$*, where*

- $N_{c_1}, .., N_{c_k}$ *are the tuples of names given to the arguments of type* `T` *at call sites* c_1 *to* c_k *of* `m()`*, and*
- N_{decl} *is the tuple of names given to the formal parameters of type* `T` *in* `m()`*'s declaration.*

5.4 Anomaly Detection

The anomaly detection leverages the extracted argument naming examples to search for anomalies in the order in which arguments are passed to a method. An anomaly is a call site of a method where arguments of the same type are named in a way that suggests a different order than the order in the source code. For instance, Figure 5.3a shows a list of naming examples for `setEndPoints()`'s `int` arguments. We refer to naming examples with N_1, N_2 etc. Example N_5 is an anomaly, because the first argument name, `low`, is similar to names used at the second position, while the second argument name, `high`, is similar to names used at the first position. Our analysis detects such anomalies and proposes a way to avoid them (here, by reversing the arguments of example N_5).

To avoid overwhelming a user of our analysis with irrelevant reports, it is important to not report every unusual argument name as an anomaly. Our analysis reports an anomaly only if changing the order of arguments makes the arguments significantly more similar to other arguments used in their respective position then using the current order. For instance, example N_2 is not an anomaly, although the name of the first argument is dissimilar to the other names of arguments used at the first position. The reason is that the second argument name of example N_2 is similar to other names at the second position; therefore, changing the argument order would not increase the overall fit of N_2 to the other naming examples.

The key idea of our analysis is that argument names used at different call sites of a method are often similar to each other. We exploit this observation to detect anomalies by comparing argument names using a string similarity metric. Such a metric returns for each pair of strings a value in the range between zero (dissimilar) and one (very similar or equal). For each argument naming example, we compute the similarity of a name used at a particular position with other names used at this position and with other names used at other positions. If a permutation of the current argument order makes the names of an example significantly more similar to the other examples than the current order, then the analysis reports an anomaly.

An alternative to using string similarity is to check whether names are equal. However, slight variations of an argument name, such as `high` and `highEP`, would make two arguments seem different although they clearly mean the same. A string similarity metric allows for quantifying the similarity of names, and thus, to also consider variations of names.

5.4.1 Algorithm

Algorithm 7 outlines our approach for detecting anomalies. The algorithm takes a list of argument naming examples as input and outputs a set of permutations that each resolve an anomaly. The algorithm iterates over all examples, and for each example, goes through all possible permutations of the example's names. The core of the algorithm are lines 6 to 15. Here, it computes a score, $permScore_{norm}$, that indicates how "normal" the argument names are with a permutation P. That is, the score expresses how similar the reordered names are to other names found at their respective positions. If a permutation of the current argument order has a score that is significantly higher than the score $currentScore$ of the current argument order, then the analysis reports an anomaly and proposes to reorder the arguments according to the permutation.

Permutations

We represent a permutation as a set of assignments of argument names to a position:

$$\begin{aligned} P &\subseteq N \times \{1, \ldots, |N|\} \\ &= \{(n, i) \mid P \text{ assigns name } n \text{ to position } i\} \end{aligned}$$

For example, the naming example N_5 as shown in Figure 5.3a is represented as:

$$\{low \mapsto 1, high \mapsto 2\}$$

Inverting the two arguments is represented as:

$$\{low \mapsto 2, high \mapsto 1\}$$

Algorithm 7 Anomaly detection based on string distance between argument names.

```
Input: Argument naming examples $\mathcal{N}$
Output: Warnings $\mathcal{W}$, each being a pair of a permutation that resolves an anomaly
    and a confidence value
 1: for all $N \in \mathcal{N}$ do
 2:     $currentScore \leftarrow 0.0$
 3:     $bestScore \leftarrow 0.0$
 4:     $P_{best} \leftarrow$ current permutation
 5:     for all $P \in permutations(N)$ do
 6:         $permScore \leftarrow 0$
 7:         for all $n \in N$ do
 8:             $posScore \leftarrow 0$
 9:             for all $i \in \{1, \ldots, |N|\}$ do
10:                 if $(n, i) \in P$ then
11:                     $posScore \leftarrow posScore + score_{assign}(n, i)$
12:                 else
13:                     $posScore \leftarrow posScore - score_{assign}(n, i)$
14:             $posScore_{norm} \leftarrow (posScore + |N| - 1)/|N|$
15:             $permScore \leftarrow permScore + posScore_{norm}$
16:         $permScore_{norm} \leftarrow permScore/|N|$
17:         if $isCurrent(P)$ then
18:             $currentScore \leftarrow permScore_{norm}$
19:         if $permScore_{norm} > bestScore$ then
20:             $bestScore \leftarrow permScore_{norm}$
21:             $P_{best} = P$
22:     $conf \leftarrow bestScore - currentScore$
23:     if $conf > t$ then
24:         $\mathcal{W} \leftarrow \mathcal{W} \cup \{(P_{best}, conf)\}$
```

Permutation Score

We compute a score $permScore$ for each permutation. This score is the sum of scores for each position of an argument. For example, the score for a permutation of two arguments is the sum of a score for the first position and a score for the second position. Since the permutation score depends on the number of equally typed arguments, we normalize it into the range $[0, 1]$ (line 16).

Position Score

The score $posScore$ for a position depends on a score $score_{assign}(n, i)$ that indicates how well a name n fits position i. The $posScore$ for a position i is the score for assigning the name to the position proposed by the permutation minus the sum of scores for all assignments of this name to other positions. That is, the assignments of a permutation influence its score positively (line 11), while all other possible assignments influence its score negatively (line 13).

Including positive and negative scores for assignments into the overall score of a position makes the algorithm more robust to cases where an argument seems to fit multiple positions. In this case, our algorithm cannot choose a single position as the most suitable, and computing a high score for any position would be misleading. If a permutation includes highly ranked assignments but also rejects other highly ranked assignments, the overall score includes high positive and high negative assignment scores that compensate for each other. Thus, the overall score expresses the uncertainty resulting from multiple apparently suitable permutations.

Similar to the permutation score, the position score depends on the number of equally typed arguments and therefore is normalized into the range $[0, 1]$ (line 14).

Assignment Score

The score $score_{assign}(n, i)$ for assigning an argument name n to a position i indicates how well a name n fits position i. To compute $score_{assign}$, we combine the string similarity between n and all other names in the naming examples of the method. At first, we compute the average similarity $simil_i^n$ of n to the arguments used elsewhere at position i:

$simil_i^n = Avg(\{simil(n, n') \mid$
n' is argument at position i in others examples$\})$

Then, we compute the average similarity $simil_{others}^n$ of n to arguments used in other examples at positions other than i:

$simil_{others}^n = Avg(\{simil(n, n') \mid$
n' is argument at position $j \neq i$ in other examples$\})$

Finally, we combine both intermediate values into the result:

$score_{assign}(n, i) = max(0, simil_i^n - simil_{others}^n)$

Subtracting $simil_{others}^n$ from $simil_i^n$ is important to adjust the result of $simil_i^n$ to the degree to which all arguments passed to the method resemble each other. The argument names of some methods vary a lot and one cannot infer any useful information from them. To deal with such cases, we subtract $simil_{others}^n$, which can be thought of as a measure for noise, from $simil_i^n$. As a result, the score for assigning n to i is normalized to the amount of knowledge we can infer from the given names, and thus, is higher if we have more confidence in the result.

Best versus Current Permutation

The last step of Algorithm 7 is to select permutations for which we know with confidence that they make the order of arguments more "normal" than the current order. While computing scores for permutations of a naming example, the algorithm stores the score of the current permutation into $currentScore$ and the maximum score over all permutation into $bestScore$. If the best score is at least t larger than the current score, then the algorithm adds the permutation P_{best}, which has the best score, to the set $\mathcal{W}$ of reported warnings. We discuss how to set the threshold t in Section 5.5.3.

The output of the algorithm a set of warnings. Each warning consists of a permutations that avoids an anomaly and a confidence value that indicates how confident the analysis is that the warning should be reported.

Ex.	Pos. 1	Pos. 2
N_1	high	low
N_2	h	Low
N_3	high	low
N_4	highEP	lowEP
N_5	**low**	**high**

(a) Arguments.

	high	highEP	h	low	lowEP	Low
high	1	0.71	0	0	0	0
highEP	0.71	1	0	0	0.60	0
h	0	0	1	0	0	0
low	0	0	0	1	0.53	1
lowEP	0	0.60	0	0.53	1	0.53
Low	0	0	0	1	0.53	1

(b) String similarities.

$$\begin{aligned}
score_{assign}(low, 1) &= max(0, 0 - 0.85) = && 0\\
score_{assign}(low, 2) &= max(0, 0.85 - 0) = && 0.85\\
score_{assign}(high, 1) &= max(0, 0.62 - 0) = && 0.62\\
score_{assign}(high, 2) &= max(0, 0 - 0.62) = && 0
\end{aligned}$$

$$\begin{aligned}
&permScore_{norm} \text{ of } \{high \mapsto 1, low \mapsto 2\}\\
=&\frac{\frac{0.85-0+2-1}{2} + \frac{0.62-0+2-1}{2}}{2} = 0.87\\
&permScore_{norm} \text{ of } \{low \mapsto 1, high \mapsto 2\}\\
=&\frac{\frac{0-0.85+2-1}{2} + \frac{0-0.62+2-1}{2}}{2} = 0.13\\
&conf \text{ of } \{high \mapsto 1, low \mapsto 2\} = 0.87 - 0.13 = 0.74
\end{aligned}$$

(c) Score computation for example N_5.

Figure 5.3: Examples of anomaly detection.

5.4.2 Example

Figure 5.3 illustrates the anomaly detection technique with an example. Figure 5.3a shows five naming examples for the method `setEndPoints()`. Suppose that N_1 has been extracted from the declaration of `setEndPoints()` and that $N_2, \ldots, N_5$ are gathered from call sites of the method. The algorithm traverses these naming examples and analyzes each permutation of the given argument names, that is, five permutations that each reverse the first and second argument of an example.

We compute the string similarities between all involved argument names (Figure 5.3b). Different string similarity metrics provide different results here. The shown numbers are computed with the *TFIDF* metric. We discuss and compare several metrics in Section 5.5.3.

The argument names of example N_5 deviate from the other naming examples. Their names suggest reversing the arguments, that is, to order them according to the permutation $\{(high, 1), (low, 2)\}$. Figure 5.3c illustrates how our algorithm computes the scores that indicate how "normal" this permutation and the current permutation are. The computation combines scores for each assignment of the permutation. For example, assigning `low` to position 2 has a score of $score_{assign}(low, 2) = 0.85$, because $simil_2^{low} = 0.85$ and $simil_{others}^{low} = 0$. The overall score for the permutation is 0.87, whereas the score for the current permutation is 0.13. That is, the confidence for inverting the arguments of N_5 is 0.74. Because the confidence is greater than our default threshold $t = 0.4$, the analysis reports a warning about N_5 and suggests inverting the two arguments.

5.4.3 Refinements

The anomaly detection presented in Algorithm 7 can be used as described so far and we show in previous work that it can effectively detect anomalies that point to real problems [126]. In the following, we describe several refinements of the approach that allow for detecting more anomalies while reporting less false positives. Some of the refinements depend on configurable parameters; we evaluate the sensitivity of the analysis to these parameters in Section 5.5.3.

Example Families

The anomaly detection described so far analyses all naming examples of a method together. Often, there are multiple common naming schemes for arguments of a method and analyzing them separately exposes more about the semantics of arguments than analyzing all together. For example, Figure 5.4a lists naming examples for `String.substring(int,int)`. Obviously (for a human), the last example is an anomaly that the analysis should report. However, some naming examples use a "start"-"end" naming scheme, whereas others use a "first"-"last" naming scheme. If the anomaly detection analyzes all naming examples together, it may miss the anomaly because permuting `end` and `start` does not have enough confidence.

We refine the approach presented so far by applying the anomaly detection to subsets of all naming examples. The subsets are chosen in such a way that they contain examples with similar names, that is, examples that are likely to use the same naming scheme. We call such subsets of naming examples *example families*.

Our algorithm for creating example families depends on a similarity measure for naming examples. Naming examples are similar to each other if their sets of argument names are similar. To compute the similarity of naming examples, we extend the string similarity measure to sets of strings as follows. Given two sets of strings $\mathcal{S}_1$ and $\mathcal{S}_2$, we find the pairwise mapping between strings from $\mathcal{S}_1$ and $\mathcal{S}_2$ where the average similarity between pairs is maximal and then report this similarity. For example, given $\mathcal{S}_1 = \{start, end\}$ and $\mathcal{S}_2 = \{startPos, endPos\}$, we find that $start \mapsto startPos$ and $end \mapsto endPos$ gives a similarity of 48%, whereas $start \mapsto endPos$ and $end \mapsto startPos$ gives a similarity of 0%. Thus, the similarity of $\mathcal{S}_1$ and $\mathcal{S}_2$ is 48%.

Based on the similarity measure for naming examples, we compute example families for each naming example observed for a method. For a particular naming example N, we perform two steps. At first, we compute the similarity of N to the other naming examples of the method. Then, we create example families that contain N and that fulfill two conditions. First, all naming examples in a family must have a similarity to N below some *coherence* value. Our implementation considers

All examples

Position 1	Position 2
start	end
startIndex	endIndex
first	last
start	end
startPos	lastPos
start	end
first	last
first	last
end	*start*

(a) Naming examples.

Coherence: 100%

Position 1	Position 2
start	end
start	end
start	end
end	*start*

Coherence: 40%

Position 1	Position 2
start	end
startIndex	endIndex
start	end
start	end
end	*start*

Coherence: 20%

Position 1	Position 2
start	end
startIndex	endIndex
start	end
startPos	lastPos
start	end
end	*start*

Coherence: 0%

Position 1	Position 2
start	end
startIndex	endIndex
first	last
start	end
startPos	lastPos
start	end
first	last
first	last
end	*start*

(b) Example families for the naming examples.

Figure 5.4: Example families.

ten coherence levels (10%, 20%, . . . , 100%), that is, we try to create families with 10% coherence, 20% coherence, etc. The rationale for trying different coherence levels is that there is no level that is works best for all programs. Second, the number of naming examples in the family must be above a configurable threshold, which we discuss in Section 5.5.3.

For the naming examples in Table 5.4a, the algorithm creates four families for the last call site in the table. These families are listed in Figure 5.4b. The first family contains only naming examples with exactly the same argument names as the last call site. The other families gradually add more and more call sites and the fourth family contains all naming examples.

The refined anomaly detection checks whether a naming example is an anomaly by analyzing each example family separately. If the analysis finds an anomaly for any of the families, then the anomaly is reported. This refined approach increases the recall of our approach compared to the approach described in [126] because the anomaly detection now detects anomalies that are obvious when analyzing a subset of naming examples but that previously have been hidden among other naming examples.

Subsets of All Parameters

The approach described so far considers all parameters of equal type at once. For methods with many equally typed parameters, this approach leads to two problems. First, an anomaly that involves only a subset of all parameters may be hidden within the other parameters. For example, consider a method with seven equally typed parameters and a call site of the method from which we extract the naming example `a,b,c,d,f,e,g`, where `f` and `e` should be inverted. If the anomaly detection considers all seven parameters, the anomaly may remain unnoticed because the difference between the scores for `a,b,c,d,f,e,g` and `a,b,c,d,e,f,g` is below the threshold. In contrast, analyzing only `f,e` and `e,f` may reveal the anomaly. Second, analyzing all permutations of naming examples of methods with many parameters is a scalability problem because each naming example of a method with k parameters has $k!$ permutations.

To address these two problems, we refine our approach by considering subsets of all equally typed parameters of a method. Instead of analyzing all parameters at once, the refined analysis considers all subsets with at least two parameters and at most four parameters. For each such subset, we run the anomaly detection separately. If for any of the subsets the analysis detects an anomaly for a call site, then this anomaly is reported to the user. If the analysis finds multiple anomalies for the same call site and argument type, then only the anomaly with the highest confidence is reported.

Considering subsets of all parameters addresses the problems describes above. First, the refined analysis discovers anomalies that otherwise would be hidden among other parameters. For the example above, the refined analysis may report that `f` and `e` should be inverted because it analyzes these two arguments without considering the other parameters of the method. Second, the refined analysis improves the scalability of the approach for methods with many equally typed parameters. Instead of considering $k!$ permutations, it considers only $2! \cdot \binom{k}{2} + 3! \cdot \binom{k}{3} + 4! \cdot \binom{k}{4}$ permutations. For example, for $k = 10$, the refinement reduces the number of permutations to consider from 3,628,800 to 5,850.

Higher Weight for Parameter Names

Parameter names should guide a developer that calls a method in assigning arguments to the correct position. To achieve this goal, parameter names often have descriptive names that convey valuable information about the semantics of the expected arguments. We found that parameter names often have more meaningful names than argument names and therefore give naming examples extracted from parameters a higher weight than naming examples extracted from arguments.

Method Filtering

The approach presented so far considers all methods with equally typed arguments. However, some methods implement commutative operations, that is, any order of arguments gives the same results. Reporting an anomaly for such a method is certainly a false positive and the analysis should not consider such methods. We observed that many methods that implement commutative operations have generic or single character parameters names, such as `op0,op1,op2` or `a,b`. The refined analysis checks for methods with such parameter names and excludes them from the analysis.

Filtering Contradicting Warnings

Some of the warnings found by Algorithm 7 may contradict each other. For example, consider two call sites `m(a,b)` and `m(b,a)` that refer to the same method `m`. When inspecting `m(a,b)` the analysis finds that changing the arguments to `m(b,a)` makes the call site more similar to all other call sites (only one in this example), and the other way around for `m(b,a)`. In general, if the analysis reports a warning for both call sites, where each warning proposes to invert the arguments, then at least one of the warnings is a false positive. We call such warnings *contradicting warnings* and remove them from the list of reported warnings. The revised analysis removes two warnings if there is a naming example N_1 with a warning that proposes P_1 and a naming example N_2 with a warning that proposes P_2, where P_1 gives N_2 and P_2 gives N_1.

5.5 Evaluation

The following section reports the results of evaluating our anomaly detection technique with real-world Java and C programs. We address the following main questions:

- *How effective is our technique in finding programming errors?* The analysis finds 31 anomalies in the Java programs, out of which 26 (84%) are relevant problems. For the C programs, the analysis find 35 anomalies, out of which 28 (80%) are relevant. To measure recall, we automatically seed bugs and our analysis finds 74% of them.
- *Which anomalies exist in mature and well-tested programs?* Eleven anomalies are correctness problems, that is, arguments that are accidentally passed in an incorrect order. Eleven other anomalies are due to badly chosen parameter names that affect the program's understandability. Finally, 32 anomalies are noteworthy because the argument order seems to be wrong but turns out to be correct.
- *How sensitive are the results on parameters of our analysis, such as the threshold for reporting anomalies?* We perform a sensitivity analysis of four parameters and discuss our default configuration.
- *Does the analysis scale to large programs?* Analyzing 2.2 millions of lines of Java and C code takes about seven minutes on a standard PC. The time required for a program correlates strongly with the number of method calls in the program.

We use the Java and C programs provided by the DaCapo 9.12 benchmark suite (Table A.2) and by the SPEC CPU 2006 benchmark suite (Table A.3). Table 5.1 lists for each program the total number of calls, the number of calls with equally typed arguments (ETA), and the number of calls with named, equally typed arguments (NETA). In total, 102,829 calls have equally typed arguments. Our analysis can extract names from 42,015 of these calls.

We use two separate techniques in this evaluation. On the one hand, we assess the effectiveness of our approach in finding real programming errors (Section 5.5.1). On the other hand, we automatically seed anomalies to assess how many of them the analysis finds (Sections 5.5.2).

Table 5.1: Programs used for the evaluation and results from the anomaly detection (ETA=equally typed arguments; NETA=named, equally typed arguments; W=warnings; CB=correctness bugs; NB=Naming bugs; NA=noteworthy anomalies).

	Program	LOC	Call sites			Anomaly Detection					
			Total	ETA	NETA	W	CB	NB	NA	Prec. (%)	Rec. (%)
Java	Avrora	69,393	20,276	3,179	878	0	0	0	0	-	73
	Batik	186,460	47,655	6,127	2,694	7	0	2	5	100	73
	DayTrader	12,325	4,613	311	103	0	0	0	0	-	81
	Eclipse	289,641	280,289	26,097	13,595	12	1	1	9	100	74
	FOP	102,909	32,806	2,796	1,266	0	0	0	0	-	76
	H2	120,821	53,221	5,210	1,607	0	0	0	0	-	72
	Jython	245,016	85,729	15,785	2,480	11	1	4	3	73	70
	Lucene	124,105	41,092	5,667	1,422	1	0	0	0	0	61
	PMD	60,062	21,394	2,601	507	0	0	0	0	-	54
	Sunflow	21,970	8,139	1,200	537	0	0	0	0	-	69
	Tomcat	161,131	54,462	4,974	1,482	0	0	0	0	-	82
	Xalan	172,300	33,828	3,663	1,650	0	0	0	0	-	76
	All	1,566,133	683,504	77,610	28,221	31	2	7	17	84	72
C	bzip2	5,731	762	110	52	0	0	0	0	-	86
	gcc	235,884	54,343	14,027	9,988	16	0	0	10	63	64
	gobmk	157,649	10,146	4,551	1,505	17	8	4	5	100	80
	h264ref	36,098	3,815	791	319	0	0	0	0	-	81
	hmmer	20,658	4,154	758	299	0	0	0	0	-	70
	lbm	904	78	9	2	0	0	0	0	-	100
	libquantum	2,606	557	176	78	0	0	0	0	-	63
	mcf	1,574	80	20	11	0	0	0	0	-	85
	milc	9,575	1,589	524	271	0	0	0	0	-	77
	perlbench	126,266	16,791	2,672	1,027	1	0	0	0	0	59
	sjeng	10,544	1,366	374	77	0	0	0	0	-	77
	sphinx3	13,128	2,952	1,207	169	1	1	0	0	-	79
	All	620,617	96,633	25,219	13,794	35	9	4	15	81	77

5.5.1 Anomalies in Mature Programs

We apply the anomaly detection to the programs listed in Table 5.1. As these programs are mature and well-tested, we do not expect to find any serious errors related to equally typed arguments. Such errors are likely to change the behavior of a program, and therefore, are typically found at some point while using the program. Nevertheless, our analysis can detect relevant anomalies that are worth the attention of programmers or maintainers, for example, to add a comment explaining an unusual piece of source code. Table 5.1 summarizes the results; Tables B.3 and B.4 give details about all reported warnings.

For the Java programs, our analysis reports 31 anomalies. We manually inspect these anomalies and classify them as follows:

- Contrary to our expectations, two anomalies are bugs affecting the program's correctness. Figure 5.1a shows the relevant source code fragments of one bug. The buggy class contains a set of public, overloaded methods that call each other and that pass multiple `int` arguments. There is an anomaly because the programmer passes the arguments in the wrong order at one call site. We were surprised to find such a bug and reported it to the Eclipse developers, who fixed it immediately (see bug 333487 in the Eclipse bug tracking system). The other bug found by the analysis is in test code of Jython. The programmer calls `assertEquals()`, which takes the actual and the expected result of some computation, and reports a warning if they are not equal. The programmer accidentally passes the expected result first, which will lead to an incorrect output if the test fails.

- Seven anomalies can be classified as *naming bugs* [74]. In these cases, the programmers chose parameter names that do not clarify the expected order of arguments. As identifier names are crucial for equally typed arguments, fixing these naming bugs would improve the understandability of the program. Figure 5.1b shows an example of a naming bug in a method computing exponentiation. The names of the two `double` parameters, `value` and `iw`, do not reveal which of the parameters refers to the base and which to the exponent. Of course, deciding about the quality of an identifier name is difficult and to some extent a matter of taste. We therefore classify only anomalies with obviously misleading names as naming bugs and count debatable cases as false positives.

- 17 anomalies can be classified as noteworthy and should be considered by the developers to improve the program's maintainability. These anomalies show unusual argument orders that seem incorrect but are intended in their specific context. For example, Eclipse has a method `resetTo(int begin, int end)` with a call site that passes arguments called `end` and `length`, which raises the question whether the arguments are ordered correctly. A closer look at the code reveals that the argument order is correct at this particular call site because the variable `end` is the index to start with. One can improve the maintainability of such source code by adding a comment explaining why a seemingly incorrect argument order is required in a particular situation.

- Finally, five anomalies are false positives. They provide no insight to a developer and, ideally, would not be reported. Most of the false positives are due to names that are similar to each other, such as two arguments called `firstName, name` where the parameter names are `name, outerFullName`. Since the analysis is based on heuristics and programmer-given identifier names, we cannot avoid false positives entirely.

For the C programs, the analysis reports 35 anomalies, which we classify as follows:

- Contrary to our expectations, nine of the anomalies are correctness bugs. Several of them involve a method `gnugo_estimate_score(float *upper, float *lower)` from gobmk. For example, one call site for which the analysis reports a warning is `gnugo_estimate_score(&lower_bound, &upper_bound)`. This code is wrong and has been fixed in a later version of the analyzed program. Another correctness bug is for method `lm_read_ctl` in sphinx3. The method takes nine parameters and three of them have type `float64`. Two of the `float64` parameters are called `wip` ("word insertion penalty") and `uw` ("unigram weight"). The method has exactly one call site, where the arguments for these two parameters are called `uw` and `inspen`, that is, they are clearly passed in the incorrect order. This bug has been fixed in a more recent of sphinx3.

- Four anomalies are naming bugs. For example, the `add_attack_move()` method of gobmk expects two `int` parameters that both represent positions. Unfortunately, the parameters are called `ww` and `pos`, making it difficult for a programmer that calls this method to distinguish the two kinds of positions.
- 15 anomalies are noteworthy. For example, Figure 5.1c is a noteworthy anomaly from gcc, where two local variables, called `if_false_label` and `if_true_label`, are passed as arguments. The arguments are ordered in such a way that `if_false_label` is bound to the parameter `if_true_label`, while `if_true_label` is bound to the formal parameter `if_false_label`. Documenting this anomaly would improve the maintainability of the code, because the natural question whether the arguments are ordered correctly does not arise.
- Seven warnings turn out to be false positives. Similar to the false positives found in the Java programs, most of them are caused by argument names that are similar to each other.

In summary, 54 of 66 reported anomalies (82%) point to problems that affect the program's correctness, understandability, or maintainability. Given that the analysis requires no input except for source code, this rate is quite satisfactory. Existing anomaly detection techniques, which search for other kinds of anomalies, often obtain lower true positive rates, for example, 29% [161], 37.5% [112], 38% [153], and 70% [74]. For a fair comparison, we use the same procedure to obtain these numbers for each work: at first, accumulate results from all programs analyzed in the respective work, and then, compute the overall true positive rate.

5.5.2 Recall

Seeding Anomalies

Measuring the recall of the anomaly detection is challenging because the set of all relevant anomalies in real-world programs is unknown. To estimate the recall, we seed anomalies in programs that are assumed to be free of problems related to equally typed arguments. By seeding anomalies, we know by construction where relevant anomalies reside, so that the evaluation is not biased by a human deciding whether a reported anomaly is relevant. This automated technique allows us to evaluate our analysis on a large scale and in an objective way.

To seed an anomaly, we take a method call site with equally typed arguments and change the order of these arguments. We then assess whether the analysis detects the seeded anomaly. We seed one anomaly after the other and run the analysis each time on the entire program. That is, we analyze a program having a single relevant anomaly and assess whether our analysis finds it. The recall for a single seeded anomaly is:

$$\text{recall} = \begin{cases} 1 & \text{if the seeded anomaly is found} \\ 0 & \text{otherwise} \end{cases}$$

The overall recall for the program is the mean value over all seeded anomalies. A similar evaluation technique has been used by others [109].

To make the results of the automated evaluation technique more meaningful and to ensure the technique's feasibility, we refine the described approach. First, we adapt the assumption that all analyzed programs are free of relevant anomalies by taking into account the known true positives described in Section 5.5.1. Since we know that these call sites expose relevant anomalies, we ignore them during the automated evaluation. Second, we ignore call sites of methods with five or more equally typed arguments for performance reasons. For a method with n equally typed arguments, we run the analysis $n! - 1$ times; thus, call sites with many arguments impose a significant performance problem. However, only around 1% of all call sites with equally typed arguments have five or more arguments, so this restriction does not affect the generality of the evaluation. Third, we apply the automated evaluation only to call sites with named arguments. For other call sites, our technique does not apply and we know without experimenting that the analysis does not report any anomalies. With these refinements, we seed 48,543 anomalies in the Java programs and 24,989 anomalies in the C programs, and run the analysis for each seeded anomaly.

Recall

Table 5.1 shows the recall of the analysis for each program. For example, the analysis finds 74.2% of all anomalies that we seed in Eclipse. On average, the analysis reveals 72% and 77% of all anomalies for Java and C, respectively. Although automatically seeded anomalies may not be representative for real-world anomalies, these results give us some confidence that the analysis finds most real anomalies. The refinements described in Section 5.4.3 significantly improve recall compared to our previously presented approach, which has only 38% recall.

5.5.3 Parameter Calibration

The presented analysis involves certain choices and parameters that have a strong influence on the overall results. We evaluate different configurations by measuring recall with seeded anomalies (Section 5.5.2) and by estimating precision. To precisely measure precision we would have to inspect for each configuration all warnings that the analysis reports. Instead, we compute a lower bound of the real precision by considering warnings to be false positives unless we know them to be true positives. In the following, we discuss the parameters and analyze the sensitivity of the analysis to each of them. We report results from varying each parameter individually while using default values for the others.

Threshold for Anomalies

The threshold for anomalies determines how deviant from other examples a call site must be to be considered an anomaly. In Algorithm 7, we call this threshold t. We experiment with values in the range between 0.1 (little deviance from other examples) and 0.9 (large deviance from other examples).

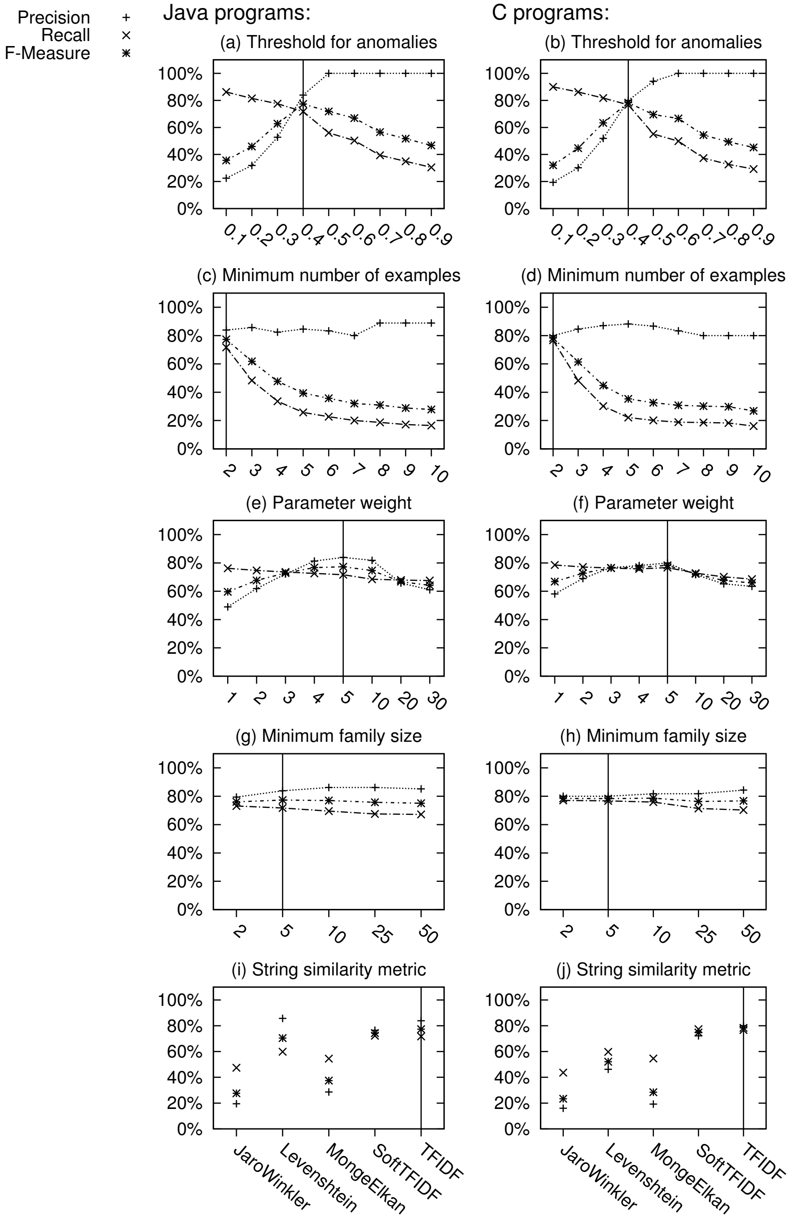


Figure 5.5: Parameters of the anomaly detection and their influence on precision and recall. The vertical lines indicate the default configuration we use for the evaluation.

Figures 5.5a and 5.5b show precision and recall with different thresholds for Java and C programs, respectively. The results illustrate the typical tradeoff between optimizing an analysis for precision and for recall. A higher threshold leads to less reported anomalies, and hence, increases precision while decreasing recall. In contrast, one can obtain a higher recall with a lower threshold for the price of losing precision. We choose a threshold of 0.4 as the default configuration, because it provides the best F-measure for Java and a close to best F-measure for C.

Minimum Number of Examples

The minimum number of examples determines how many naming examples for a method we require to draw any conclusions about the method at all. If we have fewer examples than this minimum number, our analysis ignores all call sites of the method. Note that the names of formal parameters serve as an additional naming example. We experiment with values in the range between 2 and 10.

Figures 5.5c and 5.5d show the influence of this parameter. Similarly to the threshold for anomalies, one must choose it considering the tradeoff between precision and recall. The default configuration is to require at least two naming examples. This value allows for analyzing methods with a single call site because the call site and the formal parameter names are give two naming examples.

Parameter Weight

The parameter weight determines how influential parameter names are compared to argument names. If the parameter weight is one, naming examples from parameters are considered equally important as all other naming examples. We experiments with values in the range between one and 30.

Figure 5.5e and 5.5f show how the parameter weight influences precision and recall (note the non-linear x-axis). Giving parameter weights a higher weight than argument names improves precision, which is not surprising because parameter names should guide callers of method in assigning arguments to positions. Large parameter weights decrease precision and recall, showing that considering only parameter names (and ignoring argument names altogether) does not work well. Our default configuration is a parameter weight of five, which gives the best F-measure for both Java and C.

Minimum Family Size

The minimum family size determines how many naming examples with similar names we need at least to analyze them for anomalies. We experiment with values in the range between two and 50.

Figure 5.5g and 5.5h show the results (again, note the non-linear x-axis). Compared to the other parameters, the minimum family size has a small influence on the overall results. Increasing the parameter slightly increases precision and decreases recall.

String Similarity Metric

There are various metrics to measure the similarity or distance of two strings. We experiment with five metrics, which have been found to be useful for comparing names [33].

Figures 5.5i and 5.5j compare the results obtained with the five metrics. Interestingly, choosing the string similarity metric significantly influences the overall results. Two metrics, TFIDF and SoftTFIDF, that tokenize strings before comparing them give the best F-measure. The classical Levenshtein distance, which is the minimum number of edits needed to transform a string into another, leads to a higher precision but a lower recall. Our default is to use TFIDF.

5.5.4 Performance and Scalability

On a standard PC (3.16 GHz Intel Core 2 Duo with 2 GB RAM for the Java virtual machine), our prototype implementation requires seven minutes to run the anomaly detection for all programs in Table 5.1. Most of the time is spent parsing source code. The largest Java program, Eclipse, requires 39 seconds. The largest C programs, gcc, requires 51 seconds. The running time strongly correlates with the number of call sites in a program (Pearson correlation coefficient: 93% for Java and 90% for C).

5.5.5 Threats to Validity

There a several threats to the validity of our results. First, the selection of programs we analyze may not be representative for a larger set of Java and C programs. We try to avoid selection bias by analyzing programs from two well-known benchmark suites [15, 71]. Second, the classification of anomalies into correctness bugs, naming bugs, noteworthy anomalies, and false positives may not reflect the classification that the developers of the respective programs would have done. To avoid biasing our results, we only classify an anomaly as a correctness bug if we are sure that it can lead to unexpected behavior. Furthermore, we validate our classification for a subset of anomalies by reporting them to the respective developers. However, assessing whether an anomaly is a naming bug or a noteworthy anomaly is subjective, and other developers may have other opinions. Third, the anomalies we seed to assess the recall of the analysis may not be representative for real-world anomalies. Forth, we focus the evaluation of recall on anomalies supported by the static analysis that extracts argument naming examples. For example, we do not seed anomalies that involve complex expressions as arguments.

5.6 Support for the Thesis

The analysis presented in this chapter supports the thesis that automatic program analysis allows for precisely detecting programming errors with little effort.

5.6.1 Automation

The analysis is fully automatic and requires only the program to analyze as its input. The key insight that enables this high degree of automation is to leverage implicit knowledge conveyed via identifier names. The best existing approach that addresses the problem of equally typed arguments are named arguments, which require programmers to explicitly specify argument-parameter mappings at each call site. In contrast to named arguments, our approach can analyze existing programs as they are, that is, without such specifications.

5.6.2 Precision

The warnings reported by the analysis have an acceptably high precision (82% with the default configuration). Although the analysis does not guarantee that each reported warning corresponds to a programming error, we consider the analysis to be precise enough for a deployment in practice. The analysis has several knobs, such as the threshold for anomalies, to control the precision of the reported warnings. These knobs allow developers to adapt the precision of the analysis to their needs.

5.6.3 Effort

The human effort for launching the analysis is minimal because it runs fully automatically on arbitrary programs. The human effort for inspecting the output of the analysis is low even though the analysis produces some false positives. Each warning affects two easy to inspect source code locations: a call site and a method definition. Inspecting these source code locations requires little effort, which is why we consider some false positives to be tolerable.

As we show in Section 5.5.4, the computational effort of the analysis is also very low. Furthermore, the analysis scales well to large programs because each method (and its call sites) is analyzed separately from other methods.

5.7 Limitations and Future Work

Future work may consider the following ideas:

- The recall of the analysis is bounded by the explanatory power of argument names. Short and meaningless names, such as `i` or `j`, not only confuse programmers, but also prevent our analysis from inferring the semantics of arguments. Future work may preprocess naming examples before searching for anomalies, for example, by expanding abbreviations or by mapping synonyms to a unified vocabulary [88].
- We apply the analysis to one program at a time. Many programs share APIs and the analysis may learn from one program's API usage about another program's API usage. Future work may extend our approach into a multi-project analysis.
- The analysis does not consider relations between multiple argument names at a single call site. For example, a method that transforms values into a new format may have arguments named `X` and `newX`, where `X` differs from call site to call site, but where the second argument name always prepends `new` to the first. Future work may extend our analysis to consider relations between names.
- Each analysis that produces false positives raises the question how developers react in the reported warnings. To answer this question, one may conduct a user study, for example, to assess how long developers take to decide whether a warning is relevant and how many of the warnings lead to a fix.

6 Brittle Parameter Types

In statically typed programming languages, the type system ensures that method arguments have a type expected by the callee. This check is done under the assumption that all subtypes of the declared parameter type are legal argument types [94]. Unfortunately, method parameters may have subtypes that are not expected by the callee. We call this situation *brittle parameter typing* (or simply a *brittle parameter*) because the safety guaranteed by the type system is easily breakable. A type system may find arguments given to brittle parameters to be legal, but in fact they are incorrect because the callee does not expect them.

This chapter presents a static analysis to find brittle parameters and to warn developers about unexpected arguments passed to methods with such parameters. The analysis is fully automatic because its only input is the source code or byte code of API clients. In particular, the analysis does not rely on specifications of expected argument types. The warnings reported by the analysis have a precision of 47% (in our default configuration). Since inspecting a warning involves only a single call site, we consider this precision to be acceptable in practice. Similar to the analysis from Chapter 5, the analysis presented here reports programming errors that affect the correctness, performance, and maintainability of a program.

The following section motivates the problem of brittle parameters with a real-world example. Sections 6.2 to 6.4 present our analysis and Section 6.5 reports the results from evaluating it. Finally, Sections 6.6 and 6.7 discuss how the analysis supports the thesis of this work and outlines directions of future work.

6.1 Motivation

As an example to motivate the problem of brittle parameters, consider the method `JMenu.add(Component)` from the Java Swing API (Figure 6.1). The declared parameter type `Component` has various subtypes, all of which are valid argument types according to the method declaration of `add()`. However, the API documentation states that a menu can only contain `JMenuItem`s and `JSeparator`s, that is, a subset of all compatible argument types. Adding a `Component` with another compatible type, such as `CheckBox`, is incorrect and causes undefined behavior.

Figure 6.2 illustrates a bug caused by passing an unexpected argument to the brittle parameter of `JMenu.add(Component)`. We found this problem in a real-world

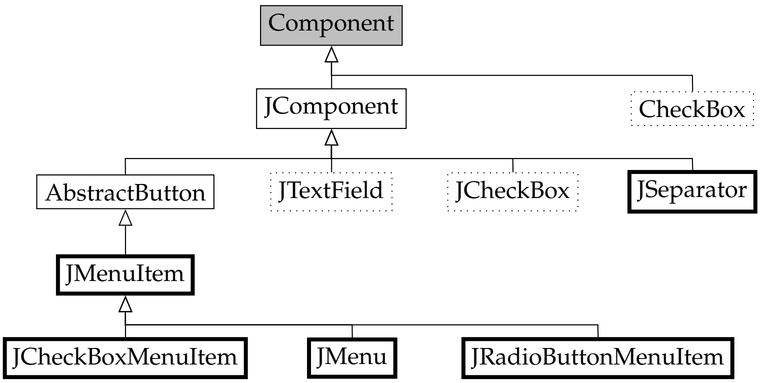


Figure 6.1: Excerpt of Swing's type hierarchy showing that `JMenu.add(Component)` has a brittle parameter. The declared parameter type has a gray background. Only the bold types should be passed as arguments. Dotted types are compatible but lead to incorrect behavior. Arguments of the remaining types may be correct or incorrect.

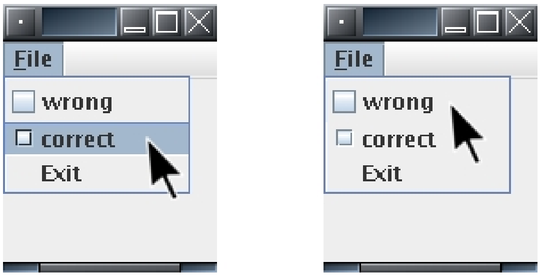


Figure 6.2: Demonstration of a real-world bug resulting from passing an unexpected argument to a brittle parameter.

program (nTorrent, a graphical user interface for a BitTorrent client[1]) and the developers confirmed it as a bug after receiving our report. The correct menu entry is of type `JCheckBoxMenuItem`. An item of this type is highlighted when hovering over it, and clicking it will close the menu. However, the programmer accidentally passes a `JCheckBox` to `JMenu.add(Component)`. In contrast to the correct menu item, the wrong item does not react on hovering, and clicking it does not give the expected behavior.

How can developers find errors caused by compatible but unexpected argument types? Finding errors related to brittle parameters is hard. First of all, traditional compilers and type systems are oblivious to the problem because the argument type

[1]`http://code.google.com/p/ntorrent/`

will type check given the declared parameter type. Furthermore, unexpected arguments may slip through traditional testing if they do not manifest through an exception. For example, the bug in Figure 6.2 does not raise an exception or any other obvious error. Instead, it leads to non-functional GUI elements as illustrated in Figure 6.2. Finally, programmers with little experience in programming against a particular API are prone to pass unexpected arguments because the knowledge about expected argument types is often scarcely or not at all documented. For the above example, the expected types are listed in `JMenu`'s class documentation but not in the method documentation of `add()`.

There are several reasons for brittle parameter typing in API methods. One reason is that the API designers had more functionality in mind when publishing the API and wanted to leave an easy way to later add this functionality without changing the method signature. Changing a parameter type to a more specific type after releasing an API is difficult, as it may break existing client code.

Brittle parameter typing also occurs when it is impossible to create a common supertype that precisely describes all expected types. If no such supertype exists, it may be possible to let all expected types implement a marker interface [16] and to use this interface as the parameter type. Unfortunately, this approach is infeasible if one or more of the expected types are declared outside of the API and therefore cannot be changed, for example, if `String` is among the expected argument types.

6.2 Overview

We present a fully automatic, static analysis to (i) find brittle parameters and (ii) reveal unexpected arguments passed to methods with such parameters. The key idea is a simple one: We leverage existing API clients to infer the argument types that an API method expects and warn developers about apparently unexpected arguments. For this purpose, the approach analyzes call sites of API methods in API clients. Figure 6.3 provides an overview of the analysis, which consists of two main steps. As input, the analysis requires the source code or byte code of API clients. The first step is a static analysis that inspects all call sites of API methods in the clients to extract information about the type of arguments passed to API methods. We call this information *argument type observations*. The second step is to search for anomalies in the argument type observations, that is, to search for argument types that are unusual with respect the other arguments passed to the parameter. Based on the assumption that most of the observations correspond to correct API usages [68, 44, 112, 161], this step identifies call sites where the observed argument type suggests an incorrect API usage. An anomaly occurs when (i) the API method has a brittle parameter, and (ii) the client passes an argument of an unexpected type.

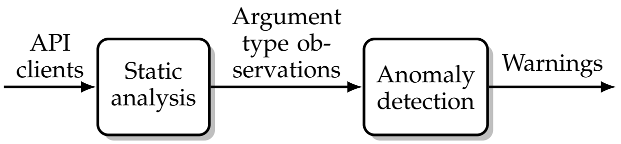


Figure 6.3: Overview of the approach.

Despite being simple, the approach is easy to apply and effective in practice. It is easy to apply because it does not require any formal specification of expected argument types. Instead, all required information is automatically extracted from existing client code. As the analysis is independent of the API implementation, it is applicable to arbitrary third-party APIs. The approach is effective because it reveals real programming errors. The price for being simple and effective is that the analysis is neither sound nor complete, that is, it might report spurious warnings and miss real errors. However, our results show both problems to be manageable in practice.

6.3 Argument Type Observations

The goal of the first analysis step is to extract information about the types of arguments that clients pass to API methods. We represent this information as argument type observations.

Definition 13 (Argument type observation)**.** *An argument type observation is a tuple* $(m_{client}, line, m_{API}, pos, type)$*, where:*

- m_{client} *is the signature of the client method that calls the API method,*
- $line$ *is the source code line of the call site of the API method,*
- m_{API} *is the signature of the called API method,*
- $pos \in \mathbb{N}$ *is the position of the argument in the list of arguments passed to* m_{API} *(starting at 1), and*
- $type$ *is a type that the argument at position* pos *may have.*

6.3.1 Points-to Analysis

To extract precise argument type observations, our analysis leverages points-to information obtained from a state of the art points-to analysis [91]. The points-to analysis statically reasons about the objects that may occur at runtime of a program and about the references that may point to each object. Potential runtime objects are represented by *abstract objects*. We access the results of the points-to analysis with a function $P2A$ that, given a reference r, returns the set $P2A(r)$ of abstract objects to which r may point. Each abstract object has an associated type. Unless specified otherwise, we use a context-insensitive points-to analysis with on the fly call graph construction. While context-sensitive analysis could provide even more precise argument type observations, we build upon a context-insensitive analysis to ensure that the approach scales to large programs.

Using a points-to analysis increases the precision of the extracted argument type observations, and therefore, the overall precision of our approach. However, the main idea of this work is independent of points-to analysis, and it can be pursued without any points-to information. In Section 6.5.7, we compare the results of our approach with and without points-to information.

6.3.2 Extraction Algorithm

Algorithm 8 summarizes how the analysis extracts argument type observations from a program. The analysis visits each method call site in the program. If the callee

Algorithm 8 Extract argument type observations from an API client.

```
Input: Program $P$ that uses API $A$
Output: Set of argument type observations $\mathcal{O}$
 1: $\mathcal{O} \leftarrow \emptyset$
 2: for all $c \in allCalls(P)$ do
 3:     if $caller(c) \in P \wedge callee(c) \in A$ then
 4:         $G \leftarrow arguments(c)$
 5:         for $pos \leftarrow 1, |G|$ do
 6:             $r \leftarrow reference(G(pos))$
 7:             if $P2A(r) \neq \emptyset$ then
 8:                 for all $o \in P2A(r)$ do
 9:                     $obs \leftarrow (caller(c), line(c), callee(c), pos,$
                            $type(o), \frac{1}{|P2A(r)|})$
10:                     $\mathcal{O} \leftarrow \mathcal{O} \cup \{obs\}$
11:             else
12:                 $obs \leftarrow (caller(c), line(c), callee(c), pos,$
                        $type(r), 1)$
13:                 $\mathcal{O} \leftarrow \mathcal{O} \cup \{obs\}$
```

of a call is an API method, it will be further analyzed (line 3). Call sites of methods defined by client types that inherit from API types are analyzed if the declaring type of the callee is an API type.

For a call site of an API method, the algorithm analyzes each of its arguments. An argument is represented by a reference (line 6), which, for example, corresponds to a local variable or a field. The analysis is performed on an intermediate program representation that has an explicit reference for each argument, even if there is no such reference in the source code. For example, the statement `m2(m1())`, which passes the return value of a call to `m1()` as an argument to `m2()`, is represented by storing the result of `m1()` into a fresh local variable, and afterwards passing this variable to `m2()`.

For each argument reference r, the analysis checks whether the points-to analysis knows any abstract objects that r may point to (line 7). If so, then the analysis creates an argument type observation for each abstract object o that r may point to. The observation states that the argument passed to the API method can have the type $type(o)$, that is, the most specific type that the points-to analysis knows for o. If r may point to multiple abstract objects, the analysis creates a separate observation for each abstract object. Doing so naively can lead to many observations for a single call site, giving this call site a higher weight than other call sites. Since we want to give each call site the same weight when analyzing the API usage of a client, we extend the definition of argument type observations by adding a confidence value:

Definition 14 (Argument type observation, extended). *An argument type observation is a tuple* $(m_{client}, line, m_{API}, pos, type, conf)$*, where:*

- m_{client} *is the signature of the client method that calls the API method,*
- $line$ *is the source code line of the call site of the API method,*
- m_{API} *is the signature of the called API method,*

```
1  class API {
2    void m(Object o, Component c) { .. }
3  }
4
5  class Client {
6    API api;
7    void n() {
8      Foo f = new Foo();
9      Component c;
10     if (..) c = new JLabel();
11     else c = new Button();
12     api.m(f, c);            // call to API method
13   }
14 }
```

Figure 6.4: Example of extracting argument type observations.

- *pos* $\in \mathbb{N}$ *is the position of the argument in the list of arguments passed to* m_{API} *(starting at 1),*
- *type is a type that the argument at position pos may have, and*
- $conf \in [0,1]$ *indicates the confidence that the argument has this type.*

When creating multiple observations for a single argument reference r that may point to different abstract objects, the analysis sets the confidence of each such observation to $\frac{1}{|P2A(r)|}$. That is, the more types the analysis observes for a single call site, the less confidence it has into each individual observation. By dividing the confidence, the analysis gives the same weight to all call sites in the program.

The points-to analysis may not know any abstract object for an argument reference r (line 11). For example, this happens if r is a parameter obtained by the caller method $caller(c)$ and if there is no known call site where $caller(c)$ is called. However, $caller(c)$ may nevertheless be called, for example, in source code that is not part of the analyzed code base, such as sub-projects of a project or external plug-ins. Therefore, we also analyze call sites where no abstract objects are known for the arguments. In this case, the analysis considers the declared type of the argument and creates an observation for this type (line 12). This observation has confidence one because the analysis makes a single observation for the call site.

6.3.3 Example

We illustrate extracting argument type observations with the simple example in Figure 6.4. Class `Client` calls an API method and passes two arguments (line 12). The analysis extracts three observations from this call site:

- $(Client.n(), 12, API.m(Object, Component), 1, Foo, 1)$
  This observation describes that the first argument for the API method has been observed to be of type `Foo`. Since this is the only possible type for this argument, the observation has confidence one.

- $(Client.n(), 12, API.m(Object, Component), 2, JLabel, 0.5)$
  This observation describes that the second argument passed to the API method has been observed to be of type `JLabel`. The local variable `c` may point to two abstract objects, which have type `JLabel` and `Button`, respectively. Therefore, the analysis splits the confidence and assigns confidence 0.5 to this observation.
- $(Client.n(), 12, API.m(Object, Component), 2, Button, 0.5)$
  This observation is similar to the previous observation, but for the argument type `Button`.

For the last two observations, the analysis relies on points-to information. Without it, the observed argument type for reference `c` is `Component` and the last two observations would be merged into a single observation with confidence one.

6.4 Detecting Anomalies

The second step of the approach, anomaly detection, has two goals. First, we want to infer from argument type observations whether a method has brittle parameters. Second, we want to reveal call sites of methods with a brittle parameter where the caller passes an argument of an unexpected type. As input, the anomaly detection takes sets of argument type observations, each obtained from a different API client. As output, it produces a set of warnings about observations of unexpected arguments. Algorithm 9 summarizes the anomaly detection.

Algorithm 9 Find anomalies in argument type observations.

```
Input: Sets of argument type observations $\mathcal{O}_1, \dots, \mathcal{O}_n$
Output: Set of warnings $\mathcal{W}$
1:  $\mathcal{O}_{raw} \leftarrow merge(\mathcal{O}_1, \dots, \mathcal{O}_n)$
2:  $\mathcal{O} \leftarrow preprocessObs(\mathcal{O}_{raw})$
3:  $M_{raw} \leftarrow param2Obs(\mathcal{O})$
4:  $M \leftarrow preprocessParams(M_{raw})$
5:  $\mathcal{W} \leftarrow \emptyset$
6:  for all $(p, \mathcal{O}_p) \in M$ do
7:      $T \leftarrow histogram(\mathcal{O}_p)$
8:      if $|\mathcal{O}_p| \geq \theta_{obs} \wedge |dom(T)| \leq \theta_{types}$ then
9:          $\mathcal{T}_{deviant} \leftarrow \emptyset$
10:         for all $t \in dom(T)$ do
11:             $conf_{incl} \leftarrow \frac{|\mathcal{O}_p|}{|dom(T)|}$
12:             $conf_{excl} \leftarrow \frac{|\mathcal{O}_p| - T(t)}{|dom(T)| - 1}$
13:             if $conf_{excl} - conf_{incl} \geq \theta_{conf}$ then
14:                 $\mathcal{T}_{deviant} \leftarrow \mathcal{T}_{deviant} \cup \{t\}$
15:         if $\frac{\sum_{t \in \mathcal{T}_{deviant}} T(t)}{\sum_{t \in dom(T)} T(t)} \leq \theta_{deviant}$ then
16:             $\mathcal{W} \leftarrow \mathcal{W} \cup \mathcal{O}_p$
```

```
1  interface APIItf1 { .. }
2  interface APIItf2 { .. }
3  interface APIItf3 { .. }
4  class API {
5    void m(APIItf1 i1) { .. }
6  }
7
8  class ClientType implements APIItf2, APIItf3 { .. }
9  class Client {
10   API api;
11   void n() {
12     api.m(new ClientType());   // call to API method
13   }
14 }
```

Figure 6.5: Example of generalizing observations by replacing client-specific types with their API supertypes.

6.4.1 Preprocessing Argument Type Observations

As a first step, the analysis merges the argument type observations from different API clients (line 1 of Algorithm 9). The $merge()$ function generalizes argument type observations to make them more comparable and then computes the union of the sets of observations. We generalize observations referring to client-specific types that are subtypes of API types. The analysis checks for each observed, client-specific argument type t_{arg} whether it has a supertype t_{API} that is defined in the API and that is compatible with the type declared by the callee. If such a supertype exists, the analysis adapts the observation by replacing t_{arg} with the supertype t_{API}. If t_{arg} has multiple most specific API supertypes, the analysis replaces the observation with a set of observations, one for each most specific API supertype. In this case, the confidence of the new observations is the confidence of the old observation divided by the number of new observations.

For example, consider Figure 6.5. The analysis extracts the following observation:

$$(Client.n(), 12, API.m(APIItf1), 1, ClientType, 1)$$

From this observation, the analysis cannot draw any conclusions about other clients because the argument type `ClientType` is client-specific. However, `ClientType` implements the API interfaces `APIItf2` and `APIItf3`, which both are compatible with the declared parameter type `APIItf1`. In this case, the analysis splits the observation into two new observations:

$$(Client.n(), 12, API.m(APIItf1), 1, APIItf2, 0.5)$$

$$(Client.n(), 12, API.m(APIItf1), 1, APIItf3, 0.5)$$

These generalized observations are useful for analyzing other API clients because they refer to API types.

After merging observations from different API clients, the analysis removes observations from which we cannot infer any information (line 2 of Algorithm 9). The

$preprocessObs()$ function removes all observations where the declared parameter type is a primitive type because primitive types have no subtypes in Java.

Next, the analysis groups observations by the declared parameter to which they refer (line 3). A declared parameter is defined by a callee and a position in the list of parameters of this callee. The map M_{raw} assigns to each parameter the set of observations made for the parameter. While considering each pair of callee and argument position as a parameter works reasonably well, we refine the notion of a parameter to consider overloaded API methods that allow clients to pass arguments of unexpected types. For example, consider the following API class:

```
class API {
  void m(Object o) { .. }
  void m(Foo o) { .. }
}
```

The overloaded method `m()` allows clients to pass any argument with a type being a subtype of `Object`. That is, from the client's perspective both methods can be considered a single method `m(Object)`. The analysis considers such cases by merging all parameters at a particular position of overloaded methods into a single parameter, if the overloaded methods have the same number of parameters. In the example, observations for both `m(Object)` and `m(Foo)` end up in a single group that refers to a parameter of type `Object`.

After grouping observations by the parameter they refer to, the analysis removes parameters where only a single argument type is observed. For such parameters, the analysis cannot reveal any unexpected arguments because either all observed arguments are expected or all observed arguments are unexpected. If all arguments are unexpected, the analysis cannot find the problem because there are no observations to learn from.

6.4.2 Type Histograms

The main part of the anomaly detection iterates over all parameters and their respective observations (line 6). For each parameter, the analysis builds a *type histogram* T showing how often each argument type is observed. T maps a type t to the summed up confidence values of all observations with argument type t.

Figure 6.6 shows four examples of type histograms extracted during the evaluation of this work. The examples illustrate kinds of histograms that occur again and again, allowing us to discuss how the anomaly detection should behave for each of them.

Figure 6.6a shows two frequently occurring types, `String` and `GridBagConstraints`, that are given to the second parameter of `Container.add(Component, Object)`. This parameter allows for specifying layout constraints for the component that is added to a container via the first argument. The expected type of the second argument depends on the layout the container uses. The layouts defined in Swing either expect a `String` describing constraints, or an instance of `GridBagConstraints`, that is, exactly the types prevalent in the histogram. The anomaly detection should not report any warnings for this histogram because all observed argument types are expected by the callee.

Figure 6.6b is the histogram for `JMenu.add(Component)`, which is discussed in Section 6.1. The declared parameter type has many subtypes, but only a small number of argument types is observed. These observed argument types correspond to the expected types specified in the API documentation (Figure 6.1). In addition to the expected types, two argument types, `JCheckBox` and `JTextField` are observed a single time each. These argument types correspond to bugs. The anomaly detection should identify these anomalies and report warnings for them.

Figure 6.6c is an example for a "long-tail" histogram, where many different types occur as arguments. The parameter type has various subtypes, and they are all expected by the method. That is, the parameter is not brittle, and therefore the analysis should not report any warnings. It is a challenge to distinguish this kind of histogram from histograms such as Figure 6.6b.

Figure 6.6d illustrates a parameter with relatively few observations. In this case, the analysis should not draw any conclusions and should report no warnings.

6.4.3 Identifying Anomalies

Given type histograms such as those in Figure 6.6, how can an analysis find unexpected arguments given to brittle parameters while reporting as few false warnings as possible?

The analysis starts with the assumption that each observation is a potential anomaly and applies four filters to remove observations that would be false warnings. An observation that passes all filters results in a warning reported to the developer. In the following, we explain the four filters. Each filter has a threshold deciding which observations to filter. In Section 6.5.6, we discuss and compare values for these thresholds.

Minimum Number of Observations The analysis ignores all parameters with a number of observations smaller than a threshold θ_{obs} (line 8 of Algorithm 9). This filter avoids drawing conclusions from a small number of observations, such as the histogram in Figure 6.6d. Setting θ_{obs} too high removes valid warnings for parameters with a fair number of observations, while choosing a value θ_{obs} that is too low leads to false warnings.

Maximum Number of Types The analysis ignores all parameters for which the number of observed argument types exceeds a threshold θ_{types} (line 8). The rationale behind this filter is that parameters for which a large number of types is observed often are not brittle, and that instead all subtypes of the declared parameter type are expected. In particular, this filter removes long-tail histograms, such as Figure 6.6c. Setting θ_{types} too low removes valid warnings, while setting it too high introduces false warnings.

Minimum Confidence Drop The analysis computes a confidence value indicating how confident it is that a given type histogram describes a brittle parameter. This confidence is the summed up confidence of all observations in the histogram divided by the number of observed argument types. If the confidence drops when adding a particular argument type to a histogram, then this argument type deviates from an otherwise accepted rule. The analysis compares the confidence with and without a type t for each type t in a histogram and ignores all types where the confidence drop is below a threshold θ_{conf} (line 13). Setting θ_{conf} too high removes valid warnings, while setting it too low leads to false positives.

Maximum Percentage of Anomalies The analysis removes all warnings for a parameter if the percentage of supposedly noteworthy observations for this parameter exceeds a threshold $\theta_{deviant}$ (line 15). The rationale for this filter is to assume that most observations correspond to correct API usage. Setting $\theta_{deviant}$ too high removes valid warnings, while setting it too low leads to false warnings.

6.4.4 Clustering Warnings

The final step of the analysis is to cluster the warnings produced by Algorithm 9 to ease their manual inspection. A cluster of warnings contains all warnings for a particular parameter. We present warnings to developers in an interactive way, where the list of warnings to inspect depends on the developer's decisions on previous warnings. Confronted with a warning, the developer can indicate whether the parameter is brittle. If so, then all calls that pass arguments to this parameter are at risk to pass an unexpected argument without a notice from the type system. In this case, the analysis presents all warnings in the cluster because they may correspond to bugs. Otherwise, the analysis knows that the cluster contains nothing but false warnings and therefore omits further warnings from the cluster.

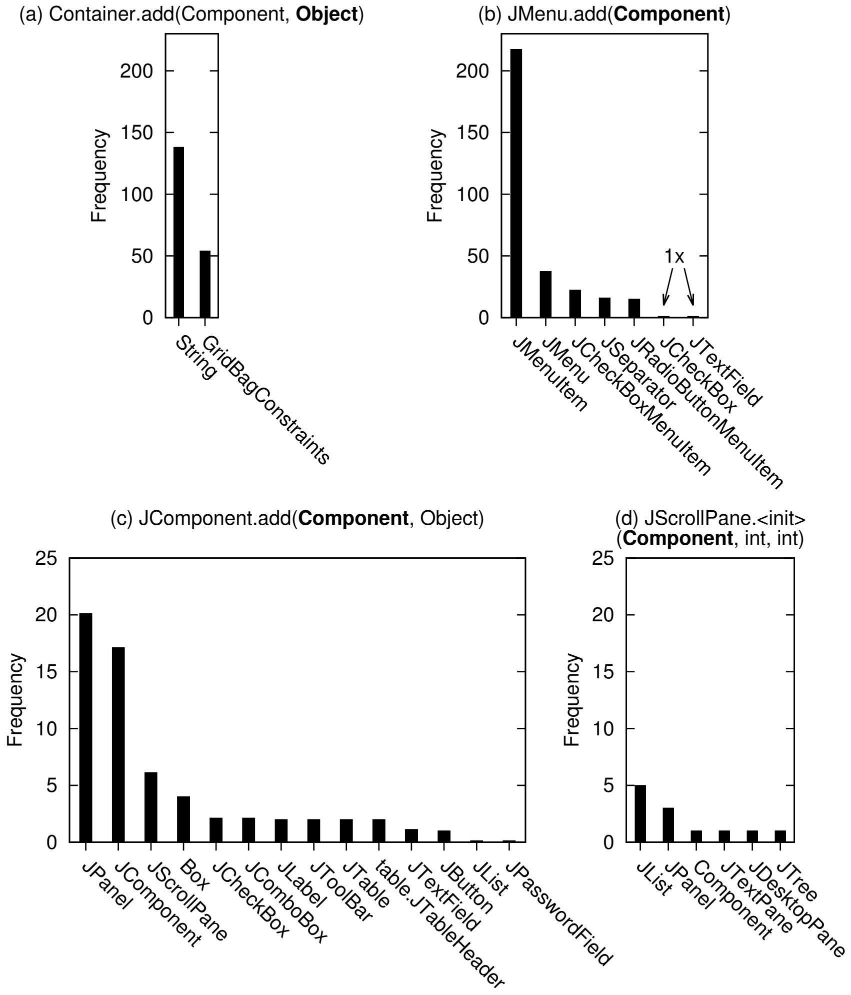


Figure 6.6: Four type histograms showing the frequency of argument types observed for the parameter printed in bold.

6.5 Evaluation

We evaluate our approach by analyzing 21 programs and their usage of the Java Swing API. The evaluation focuses on the following questions:

- *How effective is the approach in finding arguments of unexpected types?*
  The analysis reveals 15 previously unknown bugs and code smells. In its default configuration, 47% of all reported warnings point to programming errors. To measure how many unexpected arguments the analysis misses, we seed bugs into programs. The analysis detects 83% of them.
- *Does the approach scale to real-world programs?*
  Analyzing all 21 programs (650 KLoC) takes 23 minutes.
- *What is the influence of the thresholds used for detecting anomalies?*
  We run the analysis with different values for each threshold, discuss their trade-offs, and propose a default configuration.
- *What is the influence of the points-to analysis?*
  Comparing the approach with and without points-to analysis shows that points-to analysis increases precision, but that it is not crucial for the approach.

6.5.1 Implementation

We implement our approach into a practical tool for analyzing Java programs. The implementation of the static analysis is based on the Soot framework 2.5.0 [157] and its implementation of the Paddle points-to analysis [91]. We enable Soot's option to consider all methods of program classes to be reachable and exclude classes in third-party libraries from the analysis.

6.5.2 Experimental Setup and Measurements

Table A.4 lists the programs used in the evaluation. For each program, we give the number of non-comment, non-blank lines of Java source code, in total 650 KLoC. Table 6.1 shows an estimate of the number of lines of code related to the Swing API. We estimate this number by counting the source code lines of all classes that import from the `java.awt` or `javax.swing` packages. The last column of Table 6.1 shows how many argument type observations the analysis extracts from each program. In total, the analysis extracts 56,233 observations.

We inspect warnings manually and classify them into three categories [161, 65, 126]. *Bugs* are problems in the API usage that affect the correctness of the program. Since we focus on the Swing API, the bugs we find typically lead to visual glitches or they disable some functionality. *Code smells* are problems that affect performance or maintainability of the program but not its correctness. We say *true positives* to refer to both bugs and code smells. All other warnings are *false positives*.

We quantify the effectiveness of the analysis by measuring precision and recall. *Precision* means the percentage of true positives among all reported warnings. *Recall* is the percentage of true positives that the analysis reports among all true positives in the program. Since we do not know all true positives in the analyzed programs (if we had a practical way to find them, our analysis would be obsolete), we only report the

Table 6.1: Lines of API-related code and number of argument type observations.

Program	API-related lines of code	Arg. type obs.
ArgoUML 0.34	79,950	7,726
Class Editor 2.23	5,643	1,796
Dublin Core Ousia 11-01-11	3,147	583
Figoo 2.6.0	11,965	5,758
File Renamer 0.1.1	726	236
FormLayoutMaker 8.2.1rc	4,152	814
hirudo 0.7	1,569	518
id3tidy 0.3beta	829	294
jEdit 4.5pre1	67,637	7,441
JFreeChart 1.0.14	66,733	7,039
JFtp 1.53	16,180	2,958
JGraph 1.9.0.2	28,484	2,135
JPropsEdit 1.0.2	3,005	883
myPomodoro 1.0	1,960	506
nTorrent 0.5.1	4,949	901
outliner 1.8.10.6	23,434	3,499
pdfsam 2.2.1	12,197	3,933
Protégé 3.3	29,812	3,146
Scrinch 1.1.1	9,193	3,677
Stringer 1.0beta1	5,438	1,366
uBlogger 20090914	2,922	1,024
Sum	379,925	56,233

recall of seeded bugs, where we know by construction where problems in a program reside.

The anomaly detection allows for controlling the tradeoff between precision and recall with threshold parameters. In Section 6.5.6, we evaluate the influence of each threshold. For evaluating the effectiveness of the approach in finding anomalies, we use two configurations:

- *Default*. This configuration is a pragmatic compromise between maximizing true positives and minimizing false positives. The parameters are (using the notation from Algorithm 9) $\theta_{obs} = 30$, $\theta_{types} = 6$, $\theta_{conf} = 10$, and $\theta_{deviant} = 0.05$.

- *Recall-focused*. This configuration offers the possibility to reveal more true positives than the default configuration, but leads to significantly more false positives. The parameters are $\theta_{obs} = 25$, $\theta_{types} = 10$, $\theta_{conf} = 1$, and $\theta_{deviant} = 0.5$.

6.5.3 Anomalies in Real Programs

The analysis finds 15 previously unknown bugs and code smells in the programs from Table A.4. Some of them have already been fixed as a result of our bug reports. Table 6.2 shows the number of reported warnings and their classification for both

Table 6.2: Classification of warnings in real programs.

Configuration	Warnings	Bugs	Smells	True pos.
Default	19	5	4	47%
Recall-focused	155	11	4	11%

```
1  class FilteredListModel extends AbstractListModel {
2    void setFilter(String filter) {
3      Runnable runner = new Runnable() {
4        public void run() {
5          // .. update internal state ..
6
7          // BUG: first argument must be a ListModel
8          fireContentsChanged(this, 0, getSize() - 1);
9        }
10     };
11     SwingUtilities.invokeLater(runner);
12   }
13 }
```

Figure 6.7: Bug in jEdit.

configurations.[2] The default configuration gives a true positive rate of 47%, that is, about half of the reported warnings are relevant for a developer. The recall-focused configuration reveals six bugs that are not found in the default configuration. The price for finding these additional bugs is a lower true positive rate.

Figure 6.7 shows a bug that the analysis finds in jEdit. The first parameter of `fireContentsChanged()`, called in line 8, is brittle: The declared parameter type is `Object`, but the documentation clearly states that it must be a `ListModel`. Our analysis infers this constraint from call sites of this method and reports a warning because the first argument in Figure 6.7 is of type `Object`. The problem is that the programmer passes `this`, which refers to the `Runnable` and not to the surrounding class `FilteredListModel`. We reported this bug to the developers and they fixed it within a single day.[3]

Figure 6.8 is a bug found in JFtp.[4] The program wraps a list into a scroll pane to add scroll bars to it (line 6) and subsequently wraps this scroll pane into another scroll pane (line 8). The result are scroll bars surrounded by scroll bars—certainly undesired behavior. The analysis finds this problem because `JScrollPane` occurs only once as argument type of `JScrollPane`'s constructor, while several other types occur frequently.

[2]Table B.5 gives details on all reported warnings.

[3]See issue 3477759 in jEdit's bug database.

[4]See issue 3484625 in JFtp's bug database.

```java
class InsomniacClient {
  JPanel p = new JPanel();
  JList list = new JList();
  InsomniacClient() {
    // ...
    JScrollPane pane = new JScrollPane(list);
    // BUG: Two nested scroll panes
    p.add(new JScrollPane(pane));
  }
}
```

Figure 6.8: Bug in JFtp.

The analysis finds a bug in nTorrent, which is illustrated in Figure 6.2. It has been confirmed as a bug in response to our bug report.[5] Five bugs (in Protégé, jEdit, and ArgoUML) are due to using look and feel-specific color classes, which lead to visual glitches if the programs run with another look and feel than expected by the programmers. The analysis finds these bugs because the look and feel-specific classes appear as argument types instead of the much more common type `Color`. Two bugs (in JFreeChart and Scrinch) pass unexpected arguments to `Graphics2D.setClip(Shape)`. According to the documentation only particular subtypes of `Shape` will lead to the expected behavior. Another bug (in jEdit) is a `JTextField` added to a `JMenu`, which is unexpected as shown in Figure 6.1.

The four code smells found by the analysis affect performance and maintainability. Two warnings (in JPropsEdit) are about passing a newly created `JLabel` as a message to `JOptionPane.showMessageDialog()`. The API documentation states that messages are converted to `String`s by the API implementation and afterwards wrapped into a `JLabel`. A client cloning this behavior creates useless labels. The analysis warns about Protégé passing a `JComponent` to a `BoxLayout`. The underlying problem is that a class in Protégé extends `JComponent` instead of the typically extended `JPanel`. Finally, we find a problem in Scrinch, where a `StringBuffer` is passed as the message of a dialog. Since the programmer obviously does not want the API to modify the string, passing a `String` instead of the mutable `StringBuffer` would be safer.

The false positives reported by the analysis have two main reasons. First, we get false positives because some argument types that occur infrequently are nevertheless correct. For example, the analysis extracts 77 observations for `GroupLayout.setHorizontalGroup(Group)`: 76 observations with `ParallelGroup` as argument type and a single observation with `SequentialGroup` as argument type. The parameter is not brittle, that is, both argument types are expected, but the analysis cannot distinguish this case from cases like Figure 6.6b. Second, we get false positives because the static analysis extracts imprecise observations. For example, the first argument of `ActionMap.put(Object,Action)` is typically a `String`, but the analysis warns about code where an `Object` is observed. Manual inspection

[5]See issue 136 in nTorrent's bug database.

of the source code shows that the argument will be a `String` for all possible program paths, but the static analysis fails to find it.

In summary, we find that errors related to brittle parameters exist in practice and that the analysis is effective in finding them. Many of the bugs that the analysis reveals are hard to detect with traditional testing techniques. These bugs do not raise an exception or trigger any other obvious misbehavior. Instead, many bugs lead to malfunctions of the user interface or to incorrectly displayed GUI elements.

6.5.4 Automated Evaluation with Seeded Bugs

In addition to evaluating the effectiveness of the analysis in finding real bugs, we seed bugs into programs. With seeded bugs we know by construction where problems in a program reside and do not require a human to inspect warnings. Seeding bugs not only allows us to evaluate the analysis with a large number of anomalies, but also to measure the recall of the analysis.

To seed bugs related to brittle parameters, we must know about argument types that are not expected by a callee but nevertheless compatible with the declared parameter type. To find those types, we manually search the documentation of the Swing API for descriptions of brittle parameters. The result is a list of 14 API methods, each declaring a parameter of type T but expecting a proper subset of T's subtypes as argument. Based on the list B of known brittle parameters, we seed bugs into the programs from Table A.4. For each program P, we repeatedly do the following:

1. Randomly select a parameter p from B. The probability to choose parameter p is proportional to the number of call sites of p's API method in P.
2. Randomly select an argument type t from all types that are compatible with p but that are not among the expected types for p. The probability to choose a type t is proportional to the number of references with type t in P.
3. Add an argument type observation stating that t is passed to p.
4. Run the analysis to check whether it finds the seeded bug.

The seeding technique chooses API methods and argument types according their frequency in the analyzed program to simulate errors that programmers could make.

For each seeded bug, we run the analysis and measure its precision and recall:

$$\text{precision} = \frac{\#\text{true positives}}{\#\text{true positives} + \#\text{false positives}}$$

$$\text{recall} = \begin{cases} 1 & \text{if the seeded bug is found} \\ 0 & \text{otherwise} \end{cases}$$

The number of true positives is one if the analysis finds the seeded bug and zero otherwise. Based on the assumption that the programs are correct except for the seeded bug, all other reported warnings are considered to be false positives. We make this assumption more realistic by ignoring the 15 known bugs and code smells described in Section 6.5.3. Yet, the measured precision is an under-approximation because some of the warnings we count as false positives may in fact be true positives that we did not inspect manually.

We seed 100 bugs into each program from Table A.4 and compute the average of the results from all runs of the analysis. In the default configuration, the analysis has a precision and a recall of 83%. The recall-focused configuration raises recall to 94% but reduces precision to 11%. These results show that the analysis reveals most of the seeded bugs, even in the default configuration.

6.5.5 Performance

On a standard PC (Intel Core 2 Duo with 3.16 GHz, using 2.5 GB of memory), analyzing all 21 programs takes 23 minutes. Analyzing the largest program, ArgoUML, takes eight minutes. Most of the time (over 99%) is spent extracting argument type observations.

6.5.6 Thresholds of Anomaly Detection

Four thresholds control how strict the anomaly detection is when searching anomalies (Section 6.4.3). In the following, we analyze the sensitivity of the analysis to these thresholds and illustrate the tradeoffs when selecting thresholds. We vary one threshold at a time, while leaving the others at default values. Initially, we varied all four thresholds and decided on the default configuration given in Section 6.5.2.

Figure 6.9 shows how precision and recall vary depending on each threshold. For each threshold, we give the results from analyzing the original programs (left) and from analyzing programs with seeded bugs (right). For seeded bugs, we report the F-measure, that is, the harmonic mean of precision and recall.

The minimum number of observations (Figure 6.9a) controls how many observations the analysis requires to give warnings. We experiment with values between two and 200. A high threshold leads to more warnings, increasing precision but decreasing recall. In contrast, a low threshold gives higher recall but a lower precision. The figure illustrates the typical tradeoff between avoiding false positives and avoiding false negatives. Our default configuration is a compromise between both objectives.

The maximum number of types (Figure 6.9b) specifies the maximum number of different observed argument types for a parameter for which the analysis reports warnings. We experiment with values between two and 50 (Figure 6.9b shows only values up to 15 because there are no significant changes between 15 and 50). The figure shows that starting from a relatively small number (three for real bugs and four for seeded bugs), the threshold does not significantly influence the results. Our default configuration is to allow up to six different argument types.

The minimum confidence drop (Figure 6.9c) controls how much the confidence in the brittleness of a parameter must decrease by adding an argument type to a histogram for the analysis to report a warning. We experiment with values between zero and 50. A small threshold leads to more warnings, and hence, higher recall but lower precision. In contrast, a larger threshold increases precision for the price of reducing recall. As a default, we select a threshold that maximizes the F-measure for seeding bugs.

The maximum percentage of anomalies (Figure 6.9d) limits the degree of incorrectness that the analysis expects in the programs. We experiment with values between 0.1% and 100%. For very small thresholds the analysis does not report any warnings. At some point (3% for real bugs and 0.5% for seeded bugs), the analysis begins to report warnings with high precision. Further increasing the threshold reduces precision while slightly increasing recall. We choose a default threshold that gives reasonable precision.

6.5.7 Influence of Points-to Analysis

Using a points-to analysis while extracting argument type observations has a significant performance impact. While points-to analysis can have benefits as illustrated by the example in Figure 6.4, it is unclear how these benefits manifest in practice. Therefore, we evaluate our approach without points-to analysis and compare the results to the approach as described in Section 6.3. To disable points-to analysis, the approach remains as in Section 6.3 except for the $P2A$ function, which always returns an empty set. That is, we assume that the points-to analysis never knows any abstract object that a reference points to, and therefore our analysis always considers the statically declared type of arguments.

The comparison shows that the precision of finding seeded bugs decreases from 83% to 76% when abandoning points-to information. In contrast, the precision of finding real bugs and the recall of seeded bugs remain the same. We conclude that points-to analysis influences the analysis, but that it is not crucial for the overall approach. Since the performance of our approach is reasonable when using a points-to analysis, we leverage the benefits of this analysis technique.

6.5.8 Threats to Validity

We are aware of several threats to the validity of this evaluation. First, the programs and APIs we consider may not be representative for a larger set of programs and other APIs. In particular, we select the Swing API because it is known to be large and complex; other APIs may be less amenable to the analysis. Yet, we know about various other APIs with brittle parameter types (see Section 6.7) and are confident that the analysis is applicable beyond the Swing API. Second, the classification of warnings into bugs, code smells, and false positives may not reflect the opinion of other developers. To mitigate this threat, we report a subset of all true positives to the respective developers. Third, the comparison of the effectiveness of the analysis with and without points-to analysis is based on one particular points-to analysis. Other points-to analyses may give other results and we cannot conclude about the usefulness of points-to analysis in general. Finally, the (number of) bugs we seed to evaluate recall may not accurately represent any set of real-world bugs.

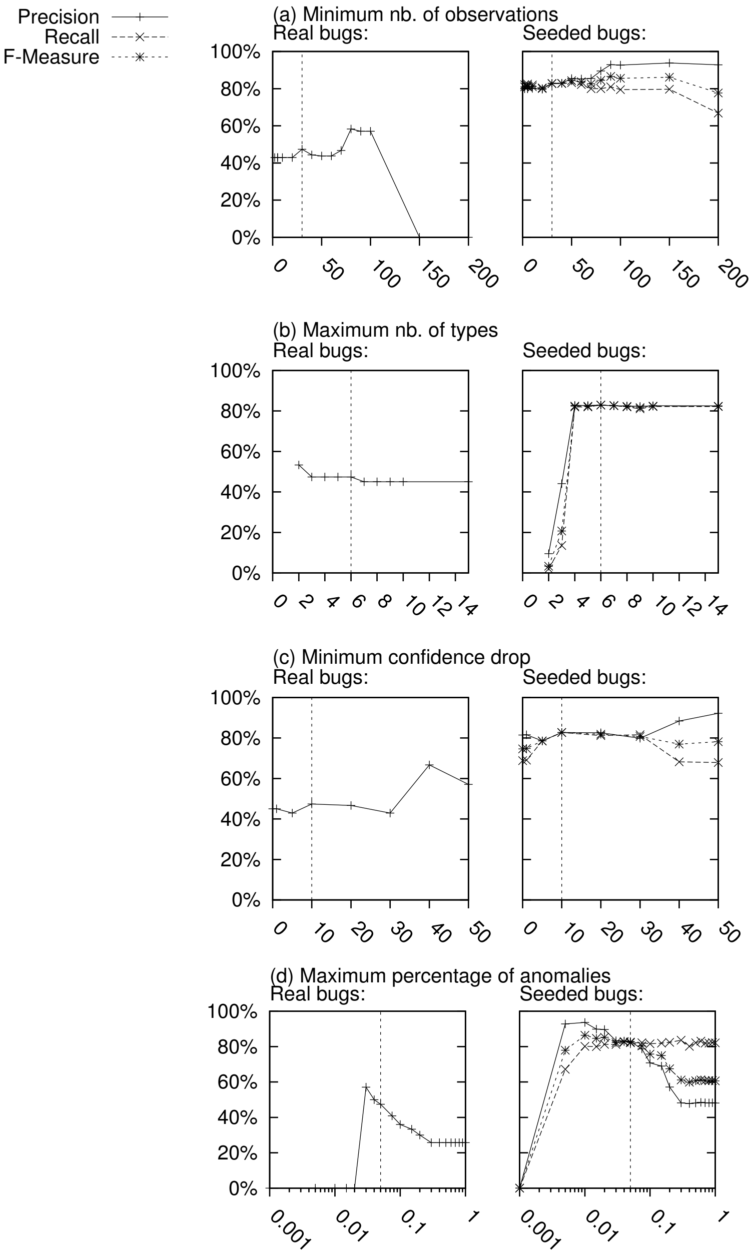


Figure 6.9: Influence of thresholds on anomaly detection. The dashed vertical line indicates the default configuration.

6.6 Support for the Thesis

The analysis presented in this chapter supports our thesis that automatic program analysis allows for precisely detecting programming errors with little effort.

6.6.1 Automation

Given a set of API clients, the analysis fully automatically identifies brittle parameters and unexpected arguments given to methods with such parameters. The analysis is independent of a particular coding style (for example, we do not rely on APIs that throw `IllegalArgumentException`s when unexpected arguments are passed) and does not rely on any kind of specification.

6.6.2 Precision

In the default configuration, 47% of the reported warnings correspond to programming errors. This precision is the lowest of the five approaches we present in this thesis. Nevertheless, we consider the precision to be acceptable for most usage scenarios because inspecting a warning takes little effort. With the recall-focuses configuration, precision drops to 11%. While this precision may be acceptable in some usage scenarios, we do not recommend the recall-focused configuration as a default.

6.6.3 Effort

The human effort for using the analysis is split among the effort to set up the analysis and the effort for inspecting the reported warnings. To set up the analysis, developers require a collection of API clients. Since brittle parameters often are the result of enhancing and revising a complex API over time, there typically are multiple clients that use an API with brittle parameters. If no client uses an API, then there is no need to enhance and revise it. To analyze an API client, it suffices to provide its source code or byte code. Therefore, the human effort for setting up the analysis is low. The effort for inspecting the warnings reported by the analysis is also low because each warning affects a single call site of an API method.

The computational effort of the analysis (Section 6.5.5) is low, given that the analysis runs without human interaction. The part of the analysis that takes most time, extracting argument type observations, can be easily parallelized by analyzing multiple programs in parallel.

6.7 Limitations and Future Work

Directions of future work include:

- *Analyze other APIs*. The problem of brittle parameter typing exists in various APIs. For example, the `Command` class in the Eclipse Platform/Core API has a method `compareTo(Object)` that expects a `Command` instance as the parameter. Another example is the Java XPath API: The `XPath` class provides a method `evaluate(String, Object)` that expects instances of `Node` as the second parameter. To adapt our analysis to an arbitrary API, it suffices to pass the appropriate API packages to Algorithm 8. Future work may apply our approach to other APIs to compare the results with ours.

- *Knowledge base for an API*. To deploy our analysis, one could extract information on brittle parameters from many API clients and persist it in a knowledge base. This knowledge base can then be used to quickly check new API clients, which, once they reach some level of maturity, can contribute to the knowledge base.

- *User study*. Similar to the analysis presented in Chapter 5, future work could evaluate through a user study how developers react on the warnings reported by the analysis. In particular, it would be interesting to study how many false positives developers are willing to tolerate for finding subtle programming errors.

Related Work 7

7.1 Rule-based Static Checkers

Rule-based static checkers are program analyses that compare the source code of a program against a set of bug patterns. These bug patterns are typically encoded by a human and delivered with the checker. Lint [81] was one of the first static checkers and “Lint-like” tool is often used as a synonym for rule-based static checkers.

Checkers by Engler et al. [44] search for violations of system-specific rules by inferring “beliefs” of programmers. For example, their approach checks for potential null dereferences, usages of a method that may fail without checking if the method failed, and access to freed memory. Their analyses are based on rule templates that are instantiated through a mixture of statistical learning from source code and manual fine tuning. For example, the checker for usages of freed memory learns which method frees memory by checking whether arguments passed to a method are used afterwards and by limiting the search space to manually specified method names, such as “free” and “deallocate”.

FindBugs [75] and PMD [2] have popularized rule-based static checking for Java. These checkers contain various correctness bug patterns, for example, to find assignments of a field to the same field or code that dereferences a null pointer. In addition to patterns to find correctness bugs, FindBugs and PMD search for instances of various bad practices, such as using the platform-specific `\n` instead of `%n` in format strings. Some of the bugs that our analyses detect can also be found with FindBugs or PMD. For instance, FindBugs searches for code that accidentally creates empty Zip files by calling `ZipOutputStream.closeEntry()` immediately after `ZipOutputStream.putNextEntry()`. This kind of protocol violation can also be detected with the analysis presented in Chapter 4.

The techniques presented in this thesis advance upon rule-based checkers in two ways:

- *Automation.* Our analyses do not rely on manually defined bug patterns. Instead, we check programs against generic correctness criteria (Chapters 2 and 3), against automatically inferred specifications (Chapter 4), or search for anomalies under the assumption that most parts of a program are correct (Chapters 5 and 6).

- *Precision.* Existing rule-based checkers report false positives. In contrast, three of the approaches presented in this thesis guarantee to report only true positives.

Several rule-based static checkers have been successfully deployed in large software companies [11, 13], showing that automatic bug detection addresses a real-world problem.

7.2 Specification Mining and Anomaly Detection

The analyses presented in this thesis do not rely on human-written specifications. Instead, the analyses in Chapters 4 and 6 infer likely API usage specifications from existing API clients under the assumption that most parts of these clients use the API correctly. This approach, coined as specification mining [9], enables specification-dependent tasks without the need to create specifications manually. One such tasks is to check programs against specifications. We refer to bug finding techniques based on inferred specifications as *anomaly detection*, because inferred specifications describe how programs usually behave and therefore violations of inferred specifications are anomalies. In the following, we discuss existing specification mining approaches and anomaly detection techniques.

7.2.1 Specification Mining

The analysis in Chapter 4 leverages a dynamic protocol miner to reveal API usage bugs. Various protocol miners have been proposed and many of them can be plugged into the approach presented in Chapter 4. Comparing protocol miners with each other is a hard problem, which we address with an evaluation framework [124] that is beyond the scope of this thesis. The two main features that distinguish our protocol miner from many others is to consider multi-object protocols and to scale well to large execution traces [125].

Dynamic Protocol Mining

Ammons et al. pioneered to mine specifications of method ordering constraints from execution traces using a probabilistic FSM learner [9]. Reiss et al. discuss space-reducing encodings of runtime data and, as a by-product, propose an algorithm for inferring FSMs from method call sequences [134]. SMArTIC is a specification mining framework that structures the mining process into trace filtering, trace clustering, probabilistic FSM learning, and merging of FSMs [97]. Adabu infers protocols by creating states based on the return values of pure methods [38].

Gabel et al. present Javert, a tool to extract protocols that involve multiple objects [54]. At first, their analysis mines traces for a set of pre-defined micro-patterns; then, it merges the micro-patterns into larger protocols. Perracotta is a specification mining tool that focuses on scalability and possibly imperfect traces [172]. Similar to [54], the approach is to search method traces for instances of particular templates, such as two events that may only occur in strictly alternating order. Instead of searching instances of pre-defined templates, our approach can identify arbitrary usage patterns.

Lorenzoli et al. [98] infer protocols that, in addition to method ordering constraints, specify invariants that should hold when taking a transition. Ghezzi et al. [58] analyze container-like classes and infer FSMs combined with graph transformation rules. This combination describes the visible state of the specified classes more fine-grained than plain FSMs. In contrast to our approach, they consider only single-object usages.

Cook et al. compare different methods for deriving FSMs that describe software development processes [34]. Despite the differing application domain, their work is related, since they also infer temporal constraints from sequences of observed events.

Static Protocol Mining

Whaley et al. infer protocols that describe all call sequences to an instance of a class that do not lead to an exception [166]. The work is based on the assumption that programmers use a particular field of a class to track the internal state of the class. The analysis identifies sets of related methods by considering those subsets of a class' methods that access a common field. Alur et al. present a generalization of that approach [7]. Their analysis searches the most general temporal interface of a class that does not lead to an unsafe valuation of the class' fields. The above analyses are conservative, that is, the produced specifications permit more call sequences than those that are actually legal. In contrast, protocols inferred with our approach summarize the behavior observed in training programs.

Wasylkowski et al. present a static analyzer that finds temporal properties, such as that *n()* can be called after *m()* [162]. As an intermediate step, their analysis constructs FSMs that focus on the use of one particular object in a method.

Other Kinds of Specifications

Beyond protocols, one can infer other kinds of specifications. Ernst et al. were among the first to derive specifications from runtime data [45, 46]. Their analysis, Daikon, extracts invariants, which possibly involve multiple variables, from program traces. Existing dynamic specification mining approaches, such as Daikon, can be fully automated by plugging them into the approach in Chapter 4.

Salah et al. infer different usage scenarios for a single class by grouping similar method invocation sequences on instances of this class into canonical sets [138]. Acharya et al. derive partial orders of API method calls from C source code [5]. Henkel et al. describe how to infer algebraic specifications for Java container classes [70]. An algebraic specification consists of axioms that describe equalities resulting from different sequences of method calls. Their approach generates such axioms automatically and tests them for validity using generated unit tests.

7.2.2 Bug Finding via Anomaly Detection

Chapters 4, 6, and 5 present bug detection techniques that infer typical usage patterns to warn about unusual parts of a program. Various other anomaly detection techniques have been proposed. Our work contributes by addressing kinds of bugs not addressed previously (Chapters 6 and 5) and by avoiding false positives (Chapter 4).

Livshits et al. propose a technique for identifying violations of application-specific programming rules with the help of software revision repositories [96]. They build upon the assumption that methods that occur in the same check-in relate to each other and apply a data mining algorithm to spot frequent method sets. A system for detecting code injection attacks by Fetzer et al. includes a component for learning typical system call sequences from method traces based on data-flow relations between calls [48].

Hangal and Lam [68] dynamically infer invariants and report violations of these invariants as potential bugs. Their paper proposes a metric for the confidence that an invariant holds, which they use to decide whether to warn about a violation of the invariant. We adapt this metric to the analysis for finding brittle parameters and use it to filter observations based on a minimum confidence drop (Section 6.4).

PR-Miner statically mines rules saying that calling a set of methods within some context implies calling another method [92]. Ramanathan et al. propose a static technique that infers temporal constraints between pairs of methods and that uses them to find bugs [133]. Chang et al. detect missing condition checks by inferring graph-based rules from source code [26]. Thummalapenta and Xie [154] target exception handling rules and how to find their violations automatically. Wasylkowski et al. [162, 161] present analyses to statically detect missing method calls. Nguyen et al. [112] and Monperrus et al. [109] learn usage patterns to find code locations where a particular call seems to be missing.

Weimer et al. present an analysis that identifies potential bugs in error handling code by searching for unusual pairs of method calls in exceptional control flow paths [164]. A static analysis by Lu et al. [99] extracts correlations between variable accesses to find unusual pieces of code that can cause inconsistent updates and concurrency bugs.

APIs with multiple clients allow for cross-project analysis, as we show in Chapter 6. Zhong et al. [178] mine API clients to derive recommendations of code snippets. In contrast to this approach, the analysis in Chapter 6 detects bugs in the analyzed client programs. Gruska et al. [65] analyze 6,000 projects with an anomaly detection technique. Scaling the analysis in Chapter 6 to a comparable number of API clients is subject to future work.

We present a static bug finding technique based on inferred protocols in [131]. In contrast to the approach presented in Chapter 4, it can analyze those parts of a program that are not covered by generated tests but reports false positives and does not address the problem of exercising programs for protocol mining.

Høst and Østvold [74] propose an analysis to detect naming bugs. They combine two analyses to check whether the implementation of a method is consistent with its name. Their approach is based on implicit knowledge about method names that has been extracted from a large corpus of programs. Some of the anomalies detected by the approach presented in Chapter 5 are also caused by inappropriate identifier names. However, our analysis addresses argument names and the order in which arguments are passed, while Høst and Østvold analyze names of methods.

7.3 Finding Bugs Related to Equal Types

Chapter 5 addresses problems related to equally typed method parameters. Several existing approaches address the inability of type systems to discern different usages of variables having the same type. In addition to other differences discussed below, our analysis differs by specifically addressing problems involving equally typed parameters.

Guo et al. [66] dynamically infer abstract types for variables of primitive types by analyzing how these variables interact, for example, through an assignment. Similarly, Hangal and Lam [69] propose a static analysis to infer dimensions that refine the type information of primitive type and string variables. The analysis in Chapter 5 differs by analyzing programmer-given identifier names instead of the interactions of values or variables. Furthermore, we use the inferred knowledge for finding anomalies in a program. In [69], dimensions inferred from a program version that is assumed to contain no errors are used to report inconsistencies introduced by later revisions of the program. In contrast, our analysis can detect inconsistencies within a single version of a program. One could combine the techniques in [66, 69] with ours by using inferred type refinements for finding problems related to equally typed arguments.

Lawall and Lo present an analysis that infers type-like groups for `int` constants by analyzing the variables with which these constants are combined [87]. Based on these groups, the analysis detects anomalies of variable-constant pairs, such as the incorrect use of a constant. Similar to the analysis in Chapter 5, Lawall and Lo address a weakness of type checkers by extracting implicit knowledge from source code. However, instead of analyzing similarities between identifier names, their approach leverages common programming idioms.

There are several approaches to explicitly refine standard types through additional information. For example, Greenfieldboyce and Foster propose adding type qualifiers to Java to express atomic properties, such as that a variable is read-only [63]. Their approach requires programmers to annotate variables with additional information, and hence, is orthogonal to an automatic analysis like the analysis in Chapter 5.

Erwig et al. define a unit system for spreadsheet languages, which derives type-like information from headers of spreadsheet tables [47]. Similar to our approach, their work leverages user-provided natural language terms to search for errors caused by inconsistencies. While Erwig et al. deal with an otherwise untyped language, our approach addresses problems caused by a too coarse-grained existing type system. Another difference is that our analysis is robust against similar but different names, whereas the analysis in [47] requires header names to match precisely.

7.4 Detecting and Avoiding Concurrency Bugs

Chapter 2 and parts of Chapter 3 present approaches to detect concurrency bugs in supposedly thread-safe classes. Table 7.1 compares the analysis presented in Chapter 2 with other techniques to find concurrency bugs based on four criteria: whether static or dynamic analysis is used, the input and output of the analysis, and the correctness criterion. The unique feature of our work is to require only a program as input, that is, to work fully automatically.

7.4.1 Data Races

Dynamic data race detectors search for unsynchronized, conflicting accesses to shared data by analyzing happens-before relations [50, 104], by checking whether the program follows a locking discipline [139, 160], or by a combination of both techniques [114]. Our approach detects data races, if they manifest through an exception or a deadlock.

Table 7.1: Comparison with existing static (S) and dynamic (D) approaches. Input: program (P), program and tests (PT), or program, tests, and specifications (PTS). Output: bugs (B) or bugs and false positives (BF). Correctness criterion: data race (DR), atomicity violations (Atom), deadlock (DL), crash, or other.

Approach	Dyn./	Input			Output		Correctness criterion				
	Stat.	P	PT	PTS	B	BF	DR	Atom	DL	Crash	Other
[139, 160, 114, 50, 104]	D	○	●	○	○	●	●	○	○	○	○
[28]	SD	○	●	○	○	●	●	○	○	○	○
[141]	D	○	●	○	●	○	●	○	○	●	○
[10, 49, 101, 67, 121, 146]	D	○	●	○	○	●	○	●	○	○	○
[51, 142]	D	○	○	●	●	○	○	●	○	○	○
[120, 86]	D	○	○	●	●	○	○	●	○	●	○
[111]	S	○	●	○	○	●	○	○	●	○	○
[83]	D	○	●	○	●	○	○	○	●	○	○
[82]	D	○	○	●	○	●	○	○	●	○	○
[175]	D	○	●	○	●	○	○	○	○	●	○
[43, 56]	D	○	○	●	●	○	○	○	○	○	●
[102, 79, 18]	D	○	●	○	○	●	○	○	○	○	●
[52, 21]	D	○	○	●	○	●	○	○	○	○	●
[174]	SD	○	●	○	○	●	○	○	○	○	●
Our work (Chapter 2)	D	●	○	○	●	○	○	○	●	●	○

7.4.2 Atomicity Violations

Analyses for finding atomicity violations rely on specifications of atomic blocks or of sets of atomic variable accesses, provided manually [51, 142, 120] or inferred heuristically [10, 49, 67]. Inferred atomic blocks causes false positives when a violation is detected for source code that is not supposed to be atomic. Our analysis detects atomicity violations, if they lead to an exception or a deadlock.

7.4.3 Deadlocks

Naik et al. [111] search deadlocks statically but require tests. Joshi et al. [82] model check a reduced program and rely on annotations of condition variables. Both analyses report false positives. Our analysis finds deadlocks triggered by generated tests.

7.4.4 Active Testing

To avoid false positives, active testing validates potential bugs by controlling the scheduler to provoke an exception or a deadlock. The approach has been applied to data races [141], atomicity violations [120, 86], deadlocks [83], and memory-related concurrency bugs [175]. Our approach shares the idea of reporting problems only if a certainly undesired situation occurs but does not rely on manually written tests.

7.4.5 Linearizability

Herlihy and Wing introduce linearizability as a correctness criterion for concurrent objects [73]. Wing and Gong test linearizability by simulating concurrent usages of an object and by comparing the observable behavior to a sequential implementation [167]. Their approach assumes to have a sequential and a concurrent implementation of the same functionality. Line-Up [18] checks the linearizability of calls but requires manually specified method parameters. Line-Up executes all linearizations before running a concurrent test, whereas our oracle analyzes linearizations only if the test fails. Elmas et al. [43] and Fonseca et al. [52] propose linearizability-based analyses that require specifications to abstract the state of a component.

7.4.6 Other Correctness Criteria

Gao et al. [56] search for typestate errors in multi-threaded programs and rely on typestate specifications. Joshi et al. [84] filter false warnings from verifying concurrent programs by using a sequential version of a program as an oracle for the concurrent program. The approach relies on formal specifications and tests. Other approaches check for violations of inferred invariants [102, 79, 143] or unusual orderings [174]. The price they pay for not relying on explicit specifications are false positives.

7.4.7 Support for Finding Concurrency Bugs

Several techniques control the thread scheduling when running a concurrent program repeatedly, for example, based on model checking [110, 35], random scheduling [140, 19], or artificial delays [42]. These techniques reduce the time to trigger a bug and could enhance the performance of our analysis.

Pugh and Ayewah [132] and Jagannath et al. [76] address the problem of manually writing concurrent unit tests. Our work is orthogonal to theirs, because we generate tests automatically.

Joshi et al. [84] propose to filter false warnings of verifiers of concurrent programs by comparing the verification results of the concurrent program to the verification results of a non-interleaved version of the program. Their work shares the idea to leverage a sequentialized version of a concurrent program to avoid false warnings. In contrast to our approach, their approach requires specifications and a test harness for the program.

Burnim et al. propose to test for semantic linearizability by checking whether programmer-given predicates evaluate to the same values in concurrent and linearized executions [22]. Assuming that programmers write such predicates, our oracle could check them in addition to focusing on concurrency-only crashes.

7.5 Test Generation

This thesis presents three dynamic analyses that are automatic because we leverage generated tests as a driver for exercising the software under test. The analyses contribute by using test generation for protocol mining and checking (Chapter 4), by generating concurrent tests (Chapter 2), and by generating generic tests that can test a subclass as well as a superclass (Chapter 3), respectively.

There are various techniques for generating tests, such as random test generation [36, 117, 30], techniques based on model checking [159], techniques based on symbolic execution [59, 169, 25], and techniques based on genetic algorithms [53, 64]. The analysis in Chapter 4 builds upon random test generation in general [36, 30] and the feedback-directed random test generator Randoop [117] in particular. The bugs found by our analysis complement the bugs reported by Randoop's built-in, generic test oracles. Our approach extends the built-in oracles with inferred protocols that reveal problems Randoop is oblivious of otherwise.

Jaygarl et al. [78] modify Randoop's random method selection by creating parameters before calling a method and by favoring methods with low coverage in the existing tests. Zheng et al. [176] propose to select methods that are likely to affect the method under test, where influence between two methods is estimated based on whether the methods access a common field. These approaches increase the coverage of methods under test, whereas the guidance technique presented in Section 4.7 intensifies the API usage of the tested program. In contrast to [78] and [176], our guidance technique considers both parameters required to call a method and methods that may influence the state of objects. A technique for regression testing by Jin et al. [80] guides Randoop towards methods that have been modified between two versions. In contrast, we guide test generation by analyzing a single version of a program.

The concurrent test generator presented in Chapter 2 is inspired by techniques to generate sequential tests, in particular by [36, 117]. In contrast to them, our test generator creates concurrent tests that exercise an object from multiple threads. In contrast to [117], our test generator does not call methods in a random order. Instead, the test generator selects methods that may provide arguments for future method calls or that may modify the state if the object under test. Integrating more elaborate test generation techniques, such as learning from observed call sequences [173], into our approach could help to detect complex bugs faster.

Ballerina [113], which was developed concurrently with the work presented in Chapter 2, generates efficient multi-threaded tests, showing that two threads, each with a single call, can trigger many concurrency bugs. Ballerina checks test executions for linearizability (similar to [18]) and therefore produces false positives.

Claessen et al. propose to test actor-based Erlang programs by generating concurrent test and by checking executions of these tests against the sequential specification of the program [32]. Their approach and the analysis in Chapter 2 share the idea of generating concurrent tests, but their test generator relies on a state-based specification of the program under test. In contrast, our generator is fully automatic.

Tillmann and Schulte propose parameterized unit tests, where all objects involved in a test are parameters to the test [156]. Their approach generates concrete tests from manually written, parameterized tests through symbolic execution. Generic tests (Section 3.3) and parameterized tests share the idea of applying the same test to different objects. In contrast to their work, the approach in Chapter 3 generates tests fully automatically and uses them to check a subclass against its superclass.

Test generation can be enhanced by mining call sequence models [155, 171, 152, 173]. Xie and Notkin [170] propose to combine test generation and specification mining for iteratively enhancing both activities. Dallmeier et al. [37] concretize this idea by generating tests that explore not yet specified sequences of a protocol and by using the result of executing these tests to refine the protocol. The main goal of these approaches is to enhance test generation with specification mining, or vice versa, whereas the approach presented in Chapter 4 combines test generation and specification mining into an automatic bug finding technique.

7.6 Substitutability

The analyses presented in Chapters 3 and 6 relate to behavioral subtyping [145, 95] and to the substitution property [94]. Both analyses address the problem that Java-like languages do not guarantee subtype instances to be semantic substitutes for supertype instances. The analysis in Chapter 3 detects subclasses that fail to ensure substitutability, whereas the analysis in Chapter 6 reveals incorrect method call sites that may result from such subclasses.[1]

There is a large body of work on verifying that a class is a behavioral subtype of another class [90, 95, 41, 137]. In contrast to the analysis presented in Chapter 3, these approaches rely on formal specifications of the behavior of subclasses and superclasses, which is not available for most real-world classes.

Offutt et al. propose a model for bugs related to inheritance and polymorphism [115]. They describe nine kinds of anomalies, such as a subclass that modifies state defined by the superclass in a way not expected by the superclass. The analysis in Chapter 3 automatically detects these kinds of problems.

America proposes an object-oriented language that distinguishes between subclassing and subtyping [8]. Most popular languages, including Java, blend these two concepts into one, giving rise to the problems revealed by the approach in Chapter 3. Taivalsaari gives a good overview of the various notions of inheritance and their respective benefits [149].

7.7 Other Testing and Debugging Techniques

McKeeman proposes to test supposedly equivalent programs, such as multiple compilers for the same language, by comparing them with each other [108]. This idea, called differential testing, has also been used to test system programs [25] and refactoring engines [40]. Srivastava et al. statically compare multiple API implementations to find missing security checks [147]. In contrast to these approaches, our analysis in Chapter 3 analyzes software at a finer level of granularity, namely at the class-level instead of the program-level. As a result, our analysis is more widely applicable: Most Java classes extend at least one other class (in addition to `Object`) [151], whereas few programs have a supposedly equivalent program to compare with. A related idea is to compare an old and a new version of a program and to warn developers about regressions [106, 80]. In contrast to these approaches, our analysis reveals problems within a single version of a program.

Similar to the analysis presented in Chapter 4, two kinds of approaches leverage the differences between passing and failing tests. The first group of approaches compares features of passing and failing tests to identify classes and methods that are likely to cause a known failure [135, 39, 103]. The second group of approaches compares passing and failing tests to generate fixes for a known problem [165, 163]. In contrast to both approaches, the analysis in Chapter 4 discovers unknown bugs

[1]Brittle parameter can also have other reasons, for example, declared parameter types that are too general.

and leverages test generation instead of relying on existing input, such as manually written tests.

The checking step of the analysis in Chapter 4 is closely related to runtime verification of protocols [6, 105, 27]. Our checker is conceptually similar to the online checker JavaMOP [27] but works on execution traces and thus avoids some of the challenges faced by JavaMOP, for example, because we know which objects interact during an execution.

7.8 Other Related Work

In Chapter 3, we use Java PathFinder [158], which unsoundly considers concurrent executions to be sequentially consistent. This limitation could cause the superclass oracle (Section 3.4) to report a false warning if a subclass crashes with sequentially consistent execution, while its superclass crashes only with a weak memory model. We did not encounter this case during our evaluation. An extension of JPF to address this problem is described in [85].

Identifier names are the subject of several studies, which generally agree on the importance of well chosen names. A study involving 100 human participants shows that expressive names are important for program understanding [89]. In particular, the study shows that single letter names impede program understanding compared to appropriate full word names. Another study shows that instances of bad naming practices correlate with poor code quality (measured in terms in FindBugs [75] warnings) [23]. The analysis presented in Chapter 5 detects poor names of multiple equally typed method parameters, that is, in a situation where meaningful names are crucial for programmers.

Related to Chapter 6, a type system to analyze “related types” addresses the problem of calls that become trivial because the actual argument type is unrelated to the formal type of the receiver [168]. Their approach requires annotating API methods, whereas the analysis in Chapter 6 infers expected argument types. Another difference is that our approach can deal with expected argument types that are scattered over the type hierarchy, that is, argument types not precisely describable with a single supertype.

Conclusions 8

This thesis set out to explore program analyses that detect programming errors. Our main hypothesis was that automatic analysis allows for precisely detecting programming errors with little effort. The key result of this work is to validate the hypothesis by presenting a set of analyses that support the thesis.

Although the analyses presented in this thesis cover different kinds of errors, programs, and analysis techniques, we are convinced that more analyses to support our thesis can and will be developed in the future. The "Future Work" sections at the end of Chapters 2 to 6 outline some directions for future analyses.

During this work, we learned a few lessons. Not all of them are new, but we believe they are worth writing down for the benefit of future research:

- *Programs contain much more information than one might think.* Programmers write programs to obtain a particular execution behavior. Therefore, one might think that programs contain exactly the information required to describe this behavior. However, our work shows that analyzing programs can reveal other useful information, such as API protocols, hints on the order of equally typed arguments, and likely specifications of the argument types that a method expects. Programmers did not intend to provide this information in their programs, but it nevertheless exists and can be leveraged to enhance the programs.

- *A program can serve as its own oracle.* Testing requires a test oracle that decides when to notify developers about a (potential) bug. Traditionally, test oracles are written by developers who know the requirements for the software under test. Our work shows that a program can serve as its own oracle: sequentially used classes serve as an oracle for concurrently used classes, superclasses serve as an oracle for subclasses, and API usage protocols followed on some execution paths serve as an oracle for other execution paths. To leverage such implicit oracles is one of the key ideas that enable our analyses to be fully automatic.

- *Many programs know more than one program.* Different programs are often assumed to achieve a particular goal, such as using an API, in the same way. Therefore, analyzing multiple programs together (for example, as described in Chapter 6) can help to identify properties and problems that may not be obvious from a single program.

- *Static analysis can be effective despite being simple.* The static analyses in Chapters 5 and 6 are rather simple compared to the state of the art in static analysis. Nevertheless, they turn out to be effective in finding programming errors.
- *Less simple static analysis is also useful.* In Chapter 6, we compare a simple variant of the analysis to a more sophisticated variant that builds upon a points-to analysis. Our results show that a more precise (and therefore less simple) analysis improves upon the results obtained with the simple variant.
- *It is possible to precisely identify bugs with imprecise (inferred) specifications.* Most approaches to infer specifications produce specifications that are imprecise and incomplete with respect to a specification that a human would come up with. One might expect (and initially, we did it) that using such imprecise specifications to find bugs inevitably leads to imprecise warnings. However, we show in Chapter 4 that imprecise, inferred protocols can help to precisely identify bugs when protocol violations are considered together with other signs of misbehavior.
- *There are kinds of bugs not yet considered by any tools.* Even though program analysis to find bugs is a long-established field of research, there are still kinds of problems that no existing analysis considers. The analyses presented in Chapters 5 and 6 show examples for this lesson learned, and we are certain that more kinds of bugs wait to be found by future program analyses.
- *Widely accepted principles may also be widely violated.* Substitutability is one of the bedrock principles of object-oriented programming and most developers are aware of it. Nevertheless, the analysis in Chapter 3 finds a surprisingly large percentage of classes in real-world software to violate this principle.
- *Even well-tested software contains bugs.* Although this lesson is certainly not new, it is worth being mentioned. Even software that is well-tested and widely used by many people contains bugs and, at some point, these bugs may pop up and lead to unpleasant surprises.

A Programs Used for Evaluation

This appendix lists programs that we use for evaluating the analyses presented in this thesis and that are not listed in the preceding chapters.

- Lines of code are counted with sloccount. Classes are counted as the number of .class files generated by javac.
- The DaCapo benchmarks do not contain source code. Instead, we downloaded the source code of each benchmark from the respective project web site.
- For the Jython benchmark of DaCapo 2006-10-MR2, the DaCapo Jar file contains some classes that are not part of the Jython source distribution. That is, the lines of code are an underapproximation.
- For the PMD benchmark of DaCapo 2006-10-MR2, the DaCapo Jar file matches the source code of PMD 3.7, but [15] refers to PMD 1.8. We count the lines of code of PMD 3.7.
- For DaCapo 9.12, there are twelve Java programs even though there are 14 benchmarks. The reason is that DayTrader is part of the tradebeans and the tradesoap benchmarks and that Lucene is part of the luindex and lusearch benchmarks.
- We omit the SPEC CPU specrand programs because they have only 49 LOC each.

Table A.1: Programs in the DaCapo 2006-10-MR2 benchmark suite [15].

Program	Classes	Lines of code
ANTLR 2.7.2	199	34,606
BLOAT 1.0	331	40,907
chart 1.0.0	481	68,791
Eclipse 3.1.2	384	34,113
FOP 0.20.5	969	43,494
HSQLDB 1.8.04	383	71,191
Jython 2.1	890	48,177
Lucene 1.9.1	316	19,271
PMD 1.8/3.7	524	38,478
Xalan 2.4.1	535	104,627
Sum	5,012	503,655

Table A.2: Programs in the DaCapo 9.12 benchmark suite [15].

Program	Classes	Lines of code
Avrora cvs-20090612	1,837	69,393
Batik 1.7	2,803	186,460
DayTrader 2.1.4	122	12,325
Eclipse 3.5.1	2,528	289,641
FOP 0.95	1,291	102,909
H2 1.2.121 (with Derby DB 10.5.3.0)	814	120,821
Jython 2.5.1	1,158	245,016
Lucene 2.4.1	1,510	124,105
PMD 4.2.5	914	60,062
Sunflow 0.07.2	244	21,970
Tomcat 6.0.20	1,603	161,131
Xalan 2.7.1	1,237	172,300
Sum	16,061	1,566,133

Table A.3: Programs in the SPEC CPU 2006 benchmark suite [71].

Program	Lines of code
bzip2	5,731
gcc	235,884
gobmk	157,649
h264ref	36,098
hmmer	20,658
lbm	904
libquantum	2,606
mcf	1,574
milc	9,575
perlbench	126,266
sjeng	10,544
sphinx3	13,128
Sum	620,617

Table A.4: Programs used to evaluate the analysis of brittle parameters (Chapter 6).

Program	Classes	Lines of code
ArgoUML 0.34	2,278	156,305
Class Editor 2.23	148	10,121
Dublin Core Ousia 11-01-11	78	6,724
Figoo 2.6.0	219	12,911
File Renamer 0.1.1	45	1,328
FormLayoutMaker 8.2.1rc	66	4,239
hirudo 0.7	48	2,642
id3tidy 0.3beta	52	2,097
jEdit 4.5pre1	1,175	103,332
JFreeChart 1.0.14	619	93,460
JFtp 1.53	167	23,511
JGraph 1.9.0.2	415	45,768
JPropsEdit 1.0.2	50	3,374
myPomodoro 1.0	112	2,510
nTorrent 0.5.1	188	36,290
outliner 1.8.10.6	468	35,407
pdfsam 2.2.1	230	15,567
Protégé 3.3	1,188	68,383
Scrinch 1.1.1	287	13,122
Stringer 1.0beta1	154	8,947
uBlogger 20090914	81	3,206
Sum	8,086	649,244

Table A.5: Analyzed classes with thread-safe superclasses.

Program	Class	Superclass
Apache Ant 1.8.1	IdentityStack	Stack
Apache Ivy 2.1.0	EncryptedProperties	Properties
Apache OpenJPA 0.9.7	FormatPreservingProperties	Properties
Apache OpenJPA 0.9.7	TypedProperties	Properties
c3p0 0.9.0	AuthMaskingProperties	Properties
Castor 1.3.1	SafeStack	Stack
Compiere 330	AdvancedRow	Vector
Cyber Garage Media Gate 1.0	ListenerList	Vector
Eclipse 3.6	IndirectList	Vector
Eclipse 3.6	NonSynchronizedVector	Vector
EMMA 2.0.5312	XProperties	Properties
GeoTools Graph 2.7-M3	IndexedStack	Stack
H2 1.1.119	SortedProperties	Properties
iText 5.0.3	LangAlt	Properties
Java PathFinder 1.0.2	ExtendedProperties	Properties
JBoss 5.1.0	PropertyMap	Properties
JGroups 2.10.0	BoundedList	ConcurrentLinked-Queue
JNI-InChI 0.5	ResolvingProperties	Properties
JSPWiki 2.8.4	CommentedProperties	Properties
Netbeans 6.9.1	NbiProperties	Properties
Omegahat SJava 0.68	FileLocator	Vector
Omegahat SJava 0.68	TrimmedProperties	Properties
RAS SDK 11.8.4.1197	Boxes	Vector
RAS SDK 11.8.4.1197	GridAreas	Vector
RAS SDK 11.8.4.1197	GroupTreeNodes	Vector
RAS SDK 11.8.4.1197	ReportPartNodes	Vector
SPEC jbb 2005	ReportProps	Properties
WebWork Core 2.2.1	LocatableProperties	Properties
Weka 3.7.2	ProtectedProperties	Properties

Thread-safe superclasses:
java.util: Vector, Stack, Timer, Properties
java.util.logging: Logger, LogManager
java.util.concurrent: ConcurrentHashMap, ArrayBlockingQueue, ConcurrentLinkedQueue, ConcurrentSkipListMap, ConcurrentSkipListSet, CopyOnWriteArrayList, CopyOnWriteArraySet, DelayQueue, LinkedBlockingDeque, LinkedBlockingQueue, PriorityBlockingQueue, SynchronousQueue
java.util.concurrent.atomic: AtomicBoolean, AtomicInteger, AtomicIntegerArray, AtomicIntegerFieldUpdater, AtomicLong, AtomicLongArray, AtomicLongFieldUpdater, AtomicMarkableReference, AtomicReference, AtomicReferenceArray, AtomicReferenceFieldUpdater, AtomicStampedReference

B Warnings Reported by Bug Detection Techniques

This appendix lists details about the warnings reported by the analyses presented in this thesis.

Table B.1: Unsafe substitutes (Chapter 3, crash oracle).

Program	Superclass	Subclass	Failure
Castor	Stack	SafeStack	ArrayIndexOutOf-BoundsException
Com. Coll.	ArrayList	FastArrayList	StackOverflowError
Com. Coll.	CompositeCollection	CompositeSet	IllegalArgument-Exception
Com. Coll.	HashMap	MultiHashMap	StackOverflowError
Com. Coll.	SequencedHashMap	LRUMap	StackOverflowError
Com. Coll.	TreeMap	FastTreeMap	ClassCastException
Compiere	Vector	AdvancedRow	ArrayIndexOutOf-BoundsException
dom4j	DefaultDocument-Type	DOMDocumentType	UnsupportedOpera-tionException
dom4j	DefaultElement	BeanElement	ClassCastException
dom4j	DefaultElement	IndexedElement	NullPointerException
dom4j	DefaultElement	UserDataElement	NullPointerException
dom4j	DefaultHandler	DOMSAXContent-Handler	NullPointerException
dom4j	DefaultHandler	SAXContentHandler	NullPointerException
dom4j	FlyweightComment	DefaultComment	InvalidXPathExcep-tion
dom4j	FlyweightEntity	DefaultEntity	InvalidXPathExcep-tion
dom4j	FlyweightEntity	DOMEntityReference	XPathException
dom4j	LeafTreeNode	BranchTreeNode	NullPointerException

Continued on next page

Table B.1 – continued from previous page

Program	Superclass	Subclass	Failure
dom4j	XMLFilterImpl	HTMLWriter	NullPointerException
dom4j	XMLFilterImpl	XMLWriter	NullPointerException
Eclipse	Vector	NonSynchronized-Vector	IndexOutOfBounds-Exception
iText	Document	PdfDocument	NullPointerException
iText	FilterOutputStream	OutputStream	NullPointerException
iText	Hashtable	DublinCoreSchema	ClassCastException
iText	Hashtable	LangAlt	ClassCastException
iText	Hashtable	PdfA1Schema	ConcurrentModifica-tionException
iText	Hashtable	PdfSchema	ClassCastException
iText	Hashtable	XmpBasicSchema	ClassCastException
iText	Hashtable	XmpMMSchema	ClassCastException
iText	Paragraph	ListItem	ClassCastException
iText	PdfPageEventHelper	FieldPositioningEvents	UnsupportedOpera-tionException
iText	Phrase	Anchor	ClassCastException
iText	Phrase	ListItem	ClassCastException
iText	Phrase	Paragraph	ClassCastException
iText	Properties	DublinCoreSchema	ClassCastException
iText	Properties	LangAlt	ClassCastException
iText	Properties	PdfA1Schema	ConcurrentModifica-tionException
iText	Properties	PdfSchema	ClassCastException
iText	Properties	XmpBasicSchema	ClassCastException
iText	Properties	XmpMMSchema	ClassCastException
iText	Rectangle	RectangleReadOnly	UnsupportedOpera-tionException
iText	VerticalPositionMark	LineSeparator	NullPointerException
JBoss	Properties	PropertyMap	StackOverflowError
OpenJPA	Properties	FormatPreserving-Properties	Deadlock

Table B.2: Unsafe API usages detected by the analysis in Chapter 4.

Program	Source location	API method
chart	RootHandler.java:124	java.util.Stack.pop()
chart	DefaultWindDataset.java:179	java.util.List.get(int)
chart	Statistics.java:206	java.util.List.get(int)
chart	JFreeChart.java:702	java.util.List.remove(java.lang.Object)
FOP	BodyAreaContainer.java:372	java.util.ArrayList.get(int)
FOP	MIFDocument.java:962	java.util.ArrayList.get(int)
FOP	AWTRenderer.java:371	java.util.Vector.removeElementAt(int)
FOP	MIFDocument.java:910	java.util.ArrayList.get(int)

Continued on next page

Table B.2 – continued from previous page

Program	Source location	API method
FOP	MIFDocument.java:922	java.util.ArrayList.get(int)
FOP	PropertyInfo.java:129	java.util.Stack.peek()
FOP	PropertyInfo.java:124	java.util.Stack.pop()
FOP	AWTRenderer.java:379	java.util.Vector.get(int)
FOP	BodyAreaContainer.java:297	java.util.ArrayList.get(int)
HSQLDB	ZaurusChoice.java:89	java.util.Vector.elementAt(int)
Jython	PySet.java:1189	java.util.Iterator.next()
Jython	InternalTables2.java:105	java.util.Iterator.remove()
Lucene	Hits.java:141	java.util.Vector.elementAt(int)
Lucene	ConjunctionScorer.java:40	java.util.LinkedList.getFirst()
Lucene	RAMOutputStream.java:82	java.util.Vector.elementAt(int)
Lucene	RAMOutputStream.java:52	java.util.Vector.elementAt(int)
Lucene	SegmentMerger.java:77	java.util.Vector.elementAt(int)
PMD	CurrentPath.java:38	java.util.LinkedList.getLast()
PMD	CurrentPath.java:42	java.util.LinkedList.getLast()
PMD	CurrentPath.java:58	java.util.LinkedList.getLast()
PMD	CyclomaticComplexity.java:148	java.util.Stack.peek()
PMD	CyclomaticComplexity.java:85	java.util.Stack.peek()
PMD	CyclomaticComplexity.java:68	java.util.Stack.peek()
PMD	DFAPanel.java:203	java.util.List.get(int)
PMD	CyclomaticComplexity.java:56	java.util.Stack.peek()
PMD	CurrentPath.java:22	java.util.LinkedList.getLast()
PMD	CurrentPath.java:46	java.util.LinkedList.getLast()
PMD	Structure.java:44	java.util.LinkedList.getLast()
PMD	CurrentPath.java:26	java.util.LinkedList.removeLast()
PMD	CurrentPath.java:63	java.util.LinkedList.getLast()
PMD	CyclomaticComplexity.java:62	java.util.Stack.peek()
PMD	CyclomaticComplexity.java:109	java.util.Stack.peek()
PMD	Structure.java:48	java.util.LinkedList.getFirst()
Xalan	StylesheetHandler.java:1550	java.util.Stack.pop()
Xalan	StylesheetHandler.java:1457	java.util.Stack.pop()
Xalan	DOMBuilder.java:375	java.util.Stack.pop()
Xalan	StylesheetHandler.java:1587	java.util.Stack.peek()
Xalan	ExpandedNameTable.java:266	java.util.Vector.elementAt(int)
Xalan	StylesheetHandler.java:1397	java.util.Stack.pop()
Xalan	StylesheetHandler.java:1298	java.util.Stack.peek()
Xalan	ExpandedNameTable.java:246	java.util.Vector.elementAt(int)
Xalan	ElemForEach.java:246	java.util.Vector.elementAt(int)
Xalan	StylesheetHandler.java:1317	java.util.Stack.pop()
Xalan	ExpandedNameTable.java:295	java.util.Vector.elementAt(int)
Xalan	Stylesheet.java:1409	java.util.Vector.elementAt(int)
Xalan	ExpandedNameTable.java:279	java.util.Vector.elementAt(int)
Xalan	ObjectArray.java:106	java.util.Vector.elementAt(int)
Xalan	Stylesheet.java:1115	java.util.Vector.elementAt(int)
Xalan	StylesheetHandler.java:1576	java.util.Stack.pop()

Continued on next page

Table B.2 – continued from previous page

Program	Source location	API method
Xalan	ObjectArray.java:77	java.util.Vector.elementAt(int)

Table B.3: Anomalies involving equally typed arguments, detected in Java programs by the analysis in Chapter 5. Classification: CB=Correctness bug, FP=False positive, NA=Noteworthy anomaly, NB=Naming bug.

Program	Source location	Called method	Class.
Batik	BackgroundRable8Bit.java:140	getViewportBounds(Graphics-Node, GraphicsNode)	NA
Batik	BackgroundRable8Bit.java:207	getBoundsRecursive(Graphics-Node, GraphicsNode)	NA
Batik	BackgroundRable8Bit.java:283	getBackground(GraphicsNode, GraphicsNode, Rectangle2D)	NA
Batik	BridgeEventSupport.java:300	dispatchMouseEvent(String, Element, Element, Point, GraphicsNodeMouseEvent, boolean)	NA
Batik	IndexImage.java:199	splitChannel(int, int, int)	NB
Batik	IndexImage.java:203	splitChannel(int, int, int)	NB
Batik	SVG12BridgeEventSupport-.java:700	dispatchMouseEvent(String, Element, Element, Point, GraphicsNodeMouseEvent, boolean, int)	NA
Eclipse	ASTConverter.java:1893	retrieveEndingSemiColon-Position(int, int)	NA
Eclipse	ASTConverter.java:2311	resetTo(int, int)	NA
Eclipse	ASTConverter.java:2355	resetTo(int, int)	NA
Eclipse	ASTConverter.java:734	retrieveEndOfRightParenthesis-Position(int, int)	FP
Eclipse	BinaryExpression.java:1439	generateOptimizedBoolean(BlockScope, CodeStream, Label, Label, boolean)	NA
Eclipse	BinaryExpression.java:1469	generateOptimizedBoolean(BlockScope, CodeStream, Label, Label, boolean)	NA
Eclipse	EqualExpression.java:279	generateOptimizedBoolean-Equal(BlockScope, CodeStream, Label, Label, boolean)	NA
Eclipse	EqualExpression.java:281	generateOptimizedNonBoolean-Equal(BlockScope, CodeStream, Label, Label, boolean)	NA
Eclipse	HierarchyResolver.java:762	subTypeOfType(Reference-Binding, ReferenceBinding)	NB

Continued on next page

Table B.3 – continued from previous page

Program	Source location	Called method	Class.
Eclipse	Scribe.java:212	createAlignment(String, int, int, int, int, boolean)	CB
Eclipse	SyncInfoTreeChangeEvent-.java:46	isParent(IResource, IResource)	NA
Eclipse	UnaryExpression.java:185	generateOptimizedBoolean(BlockScope, CodeStream, Label, Label, boolean)	NA
Jython	cPickle.java:802	isSubClass(PyObject, PyObject)	NB
Jython	imp.java:635	find_module(String, String, PyList)	FP
Jython	imp.java:746	import_next(PyObject, String-Builder, String, String, PyObject)	FP
Jython	imp.java:756	import_first(String, String-Builder, String, PyObject)	FP
Jython	ModjyTestWSGIStreams-.java:53	assertEquals(String, int, int)	CB
Jython	PyException.java:118	isSubClass(PyObject, PyObject)	NB
Jython	PyFloat.java:520	_pow(double, double, PyObject)	NB
Jython	PyInteger.java:461	_pow(int, int, PyObject, PyObject, PyObject)	NB
Jython	TextIOWrapper.java:129	readLoop(byte[], int, char[], int, int)	NA
Jython	UniversalIOWrapper.java:123	readLoop(byte[], int, char[], int, int)	NA
Jython	zxJDBC.java:190	__setitem__(PyObject, PyObject)	NA
Lucene	TestSnowball.java:38	assertEquals(String, String)	FP

Table B.4: Anomalies involving equally typed arguments, detected in C programs by the analysis in Chapter 5. Classification: CB=Correctness bug, FP=False positive, NA=Noteworthy anomaly, NB=Naming bug.

Program	Source location	Called method	Class.
gcc	expr.c:9538	do_jump_by_parts_equality(tree_node *, rtx_def *, rtx_def *)	NA
gcc	sched-rgn.c:1133	sbitmap_copy(simple_bitmap_def *, simple_bitmap_def *)	NA
gcc	gcse.c:6500	true_dependence(rtx_def *, enum machine_mode, rtx_def *, int (*)(rtx_def *, int))	FP
gcc	expr.c:9575	do_jump_by_parts_greater(tree_node *, int, rtx_def *, rtx_def *)	NA
gcc	df.c:2835	df_pattern_emit_before(df *, rtx_def *, basic_block_def *, rtx_def *)	FP

Continued on next page

Table B.4 – continued from previous page

Program	Source location	Called method	Class.
gcc	sched-rgn.c:2342	add_dependence(rtx_def *, rtx_def *, enum reg_note)	NA
gcc	c-typeck.c:1564	convert_arguments(tree_node *, tree_node *, tree_node *, tree_node *)	FP
gcc	stmt.c:5580	case_tree2list(case_node *, case_node *)	FP
gcc	expr.c:9438	do_jump(tree_node *, rtx_def *, rtx_def *)	NA
gcc	expr.c:9494	do_jump(tree_node *, rtx_def *, rtx_def *)	NA
gcc	expr.c:9603	do_compare_and_jump(tree_node *, enum rtx_code, enum rtx_code, rtx_def *, rtx_def *)	NA
gcc	expr.c:9368	do_jump(tree_node *, rtx_def *, rtx_def *)	NA
gcc	expr.c:9162	expand_binop(enum machine_mode, optab *, rtx_def *, rtx_def *, rtx_def *, int, enum optab_methods)	FP
gcc	splay-tree.c:109	splay_tree_splay_helper(splay_tree_s *, unsigned long int, splay_tree_node_s * *, splay_tree_node_s * *, splay_tree_node_s * *)	FP
gcc	sched-rgn.c:2379	add_dependence(rtx_def *, rtx_def *, enum reg_note)	NA
gcc	expr.c:9557	do_jump_by_parts_greater(tree_node *, int, rtx_def *, rtx_def *)	NA
perlbench	win32.c:3147	rename(const char *, const char *)	FP
gobmk	sgfdecide.c:345	owl_analyze_semeai(int, int, int *, int *, int *, int)	NA
gobmk	owl.c:3670	add_owl_prevent_threat_move(int, int)	CB
gobmk	play_solo.c:228	gnugo_estimate_score(float *, float *)	CB
gobmk	genmove.c:342	estimate_score(float *, float *)	CB
gobmk	sgfdecide.c:391	owl_analyze_semeai(int, int, int *, int *, int *, int)	NA
gobmk	interface.c:322	estimate_score(float *, float *)	CB
gobmk	patterns.c:11106	defend_against(int, int, int)	CB
gobmk	worm.c:1225	add_attack_threat_move(int, int, int)	NB
gobmk	worm.c:1233	add_defense_move(int, int, int)	NB
gobmk	aftermath.c:808	estimate_score(float *, float *)	CB
gobmk	owl.c:898	do_owl_analyze_semeai(int, int, local_owl_data *, local_owl_data *, int, int *, int *, int *, int, int)	NA
gobmk	owl.c:844	do_owl_analyze_semeai(int, int, local_owl_data *, local_owl_data *, int, int *, int *, int *, int, int)	NA
gobmk	reading.c:661	break_through_helper(int, int, int, int, int, int, int, int)	NA
gobmk	play_solo.c:116	gnugo_estimate_score(float *, float *)	CB
gobmk	play_ascii.c:449	gnugo_estimate_score(float *, float *)	CB
gobmk	worm.c:1222	add_attack_move(int, int, int)	NB
gobmk	worm.c:1237	add_defense_threat_move(int, int, int)	NB

Continued on next page

Table B.4 – continued from previous page

Program	Source location	Called method	Class.
sphinx3	kbcore.c:163	lm_read_ctl(char *, dict_t *, double, double, double, char *, int *, int *, int)	CB

Table B.5: Anomalies related to brittle parameters (Chapter 6, default configuration). The "Pos." column gives the zero-based position of the unusual argument. Classification: B=Bug, CS=Code smell, FP=False positive.

Program	Source location	Called method	Pos.	Class.
ArgoUML	DnDExplorerTree.java-:276	GradientPaint(float, float, Color, float, float, Color)	2	B
ArgoUML	FigEdgeModelElement-.java:352	JMenu.add(Action)	0	FP
ArgoUML	JLinkButton.java:138	JButton.addActionListener(ActionListener)	0	FP
ArgoUML	UMLCheckBox2.java:80	JCheckBox.addAction-Listener(ActionListener)	0	FP
jEdit	ActionBar.java:47	BoxLayout(Container, int)	0	FP
jEdit	SearchBar.java:48	BoxLayout(Container, int)	0	FP
jEdit	TextAreaBorder.java:37	Graphics.setColor(Color)	0	B
jEdit	TextAreaBorder.java:40	Graphics.setColor(Color)	0	B
jEdit	TextAreaBorder.java:44	Graphics.setColor(Color)	0	B
JFreeChart	ChartPanel.java:2776	JOptionPane.showMessage-Dialog(Component, Object)	1	FP
JPropsEdit	JPropsEditOpenAction-.java:95	JOptionPane.showMessage-Dialog(Component, Object, String, int)	1	CS
JPropsEdit	JPropsEditSystem-Properties-Action.java:76	JOptionPane.showMessage-Dialog(Component, Object, String, int)	1	CS
pdfsam	JPdfSelectionPanel-.java:460	ActionMap.put(Object, Action)	0	FP
pdfsam	MixMainGUI.java:178	GroupLayout.setHorizontal-Group(Group)	0	FP
pdfsam	MixMainGUI.java:184	GroupLayout.setVertical-Group(Group)	0	FP
Protégé	ChangeProjectFormat-WizardPage.java:53	BoxLayout(Container, int)	0	CS
Protégé	ConfigureButtonIcon-.java:44	Graphics.setColor(Color)	0	B
Scrinch	PreferencesDialog.java-:100	JOptionPane.showConfirm-Dialog(Component, Object, String, int)	1	CS
Stringer	StatusBar.java:24	JPanel.add(Component, Object)	1	FP

Bibliography

[1] AspectJ. `http://www.eclipse.org/aspectj`.

[2] PMD. `http://pmd.sourceforge.net`.

[3] IEEE standard glossary of software engineering terminology, 1990.

[4] API documentation of *java.lang.StringBuffer* (Java platform standard edition 6), 2011.

[5] M. Acharya, T. Xie, J. Pei, and J. Xu. Mining API patterns as partial orders from source code: From usage scenarios to specifications. In *European Software Engineering Conference and Symposium on Foundations of Software Engineering (ES-EC/FSE)*, pages 25–34. ACM, 2007.

[6] C. Allan, P. Avgustinov, A. S. Christensen, L. Hendren, S. Kuzins, O. Lhoták, O. de Moor, D. Sereni, G. Sittampalam, and J. Tibble. Adding trace matching with free variables to AspectJ. In *Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 345–364. ACM, 2005.

[7] R. Alur, P. Cerný, P. Madhusudan, and W. Nam. Synthesis of interface specifications for Java classes. In *Symposium on Principles of Programming Languages (POPL)*, pages 98–109. ACM, 2005.

[8] P. America. Designing an object-oriented programming language with behavioural subtyping. In *REX Workshop*, volume 489, pages 60–90. Springer, 1990.

[9] G. Ammons, R. Bodík, and J. R. Larus. Mining specifications. In *Symposium on Principles of Programming Languages (POPL)*, pages 4–16. ACM, 2002.

[10] C. Artho, K. Havelund, and A. Biere. High-level data races. *Software Testing, Verification and Reliability*, 13(4):207–227, 2003.

[11] N. Ayewah, D. Hovemeyer, J. D. Morgenthaler, J. Penix, and W. Pugh. Using static analysis to find bugs. *IEEE Software*, 25(5):22–29, 2008.

[12] N. E. Beckman, D. Kim, and J. Aldrich. An empirical study of object protocols in the wild. In *European Conference on Object-Oriented Programming (ECOOP)*, pages 2–26, 2011.

[13] A. Bessey, K. Block, B. Chelf, A. Chou, B. Fulton, S. Hallem, C. Henri-Gros, A. Kamsky, S. McPeak, and D. R. Engler. A few billion lines of code later: Using static analysis to find bugs in the real world. *Communications of the ACM*, 53(2):66–75, 2010.

[14] S. M. Blackburn et al. The DaCapo benchmarks: Java benchmarking development and analysis. In *Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 169–190. ACM, 2006.

[15] S. M. Blackburn, R. Garner, C. Hoffmann, A. M. Khan, K. S. McKinley, R. Bentzur, A. Diwan, D. Feinberg, D. Frampton, S. Z. Guyer, M. Hirzel, A. L. Hosking, M. Jump, H. B. Lee, J. E. B. Moss, A. Phansalkar, D. Stefanovic, T. VanDrunen, D. von Dincklage, and B. Wiedermann. The DaCapo benchmarks: Java benchmarking development and analysis. In *Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 169–190. ACM, 2006.

[16] J. Bloch. *Effective Java (Second Edition)*. Addison-Wesley, 2008.

[17] F. P. Brooks. No silver bullet: Essence and accidents of software engineering. *IEEE Computer*, 20(4):10–19, 1987.

[18] S. Burckhardt, C. Dern, M. Musuvathi, and R. Tan. Line-Up: a complete and automatic linearizability checker. In *Conference on Programming Language Design and Implementation (PLDI)*, pages 330–340. ACM, 2010.

[19] S. Burckhardt, P. Kothari, M. Musuvathi, and S. Nagarakatte. A randomized scheduler with probabilistic guarantees of finding bugs. In *Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 167–178, 2010.

[20] S. Burckhardt and R. Tan. Private communication, August 2011.

[21] J. Burnim, T. Elmas, G. C. Necula, and K. Sen. NDSeq: runtime checking for nondeterministic sequential specifications of parallel correctness. In *Conference on Programming Language Design and Implementation (PLDI)*, pages 401–414. ACM, 2011.

[22] J. Burnim, G. C. Necula, and K. Sen. Specifying and checking semantic atomicity for multithreaded programs. In *Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 79–90. ACM, 2011.

[23] S. Butler, M. Wermelinger, Y. Yu, and H. Sharp. Relating identifier naming flaws and code quality: An empirical study. In *Working Conference on Reverse Engineering (WCRE)*, pages 31–35. IEEE, 2009.

[24] S. Butler, M. Wermelinger, Y. Yu, and H. Sharp. Exploring the influence of identifier names on code quality: An empirical study. In *European Conference on Software Maintenance and Reengineering (CSMR)*, pages 156–165. IEEE, 2010.

[25] C. Cadar, D. Dunbar, and D. R. Engler. KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *Symposium on Operating Systems Design and Implementation (OSDI)*, pages 209–224. USENIX, 2008.

[26] R.-Y. Chang, A. Podgurski, and J. Yang. Finding what's not there: a new approach to revealing neglected conditions in software. In *International Symposium on Software Testing and Analysis (ISSTA)*, pages 163–173. ACM, 2007.

[27] F. Chen and G. Rosu. MOP: An efficient and generic runtime verification framework. In *Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 569–588. ACM, 2007.

[28] J.-D. Choi, K. Lee, A. Loginov, R. O'Callahan, V. Sarkar, and M. Sridharan. Efficient and precise datarace detection for multithreaded object-oriented programs. In *Conference on Programming Language Design and Implementation (PLDI)*, pages 258–269, 2002.

[29] A. Chou, J. Yang, B. Chelf, S. Hallem, and D. R. Engler. An empirical study of operating system errors. In *Symposium on Operating Systems Principles (SOSP)*, pages 73–88, 2001.

[30] I. Ciupa, A. Leitner, M. Oriol, and B. Meyer. ARTOO: adaptive random testing for object-oriented software. In *International Conference on Software Engineering (ICSE)*, pages 71–80. ACM, 2008.

[31] I. Ciupa, A. Pretschner, A. Leitner, M. Oriol, and B. Meyer. On the predictability of random tests for object-oriented software. In *International Conference on Software Testing, Verification, and Validation (ICST)*, pages 72–81. IEEE Computer Society, 2008.

[32] K. Claessen, M. Palka, N. Smallbone, J. Hughes, H. Svensson, T. Arts, and U. T. Wiger. Finding race conditions in Erlang with QuickCheck and PULSE. In *International Conference on Functional programming (ICFP)*, pages 149–160. ACM, 2009.

[33] W. W. Cohen, P. D. Ravikumar, and S. E. Fienberg. A comparison of string distance metrics for name-matching tasks. In *Workshop on Information Integration on the Web (IIWeb)*, pages 73–78, 2003.

[34] J. E. Cook and A. L. Wolf. Discovering models of software processes from event-based data. *ACM Transactions on Software Engineering and Methodology*, 7(3):215–249, 1998.

[35] K. E. Coons, S. Burckhardt, and M. Musuvathi. GAMBIT: effective unit testing for concurrency libraries. In *Symposium on Principles and Practice of Parallel Programming, (PPOPP)*, pages 15–24. ACM, 2010.

[36] C. Csallner and Y. Smaragdakis. JCrasher: an automatic robustness tester for Java. *Software Practice and Experience*, 34(11):1025–1050, 2004.

[37] V. Dallmeier, N. Knopp, C. Mallon, S. Hack, and A. Zeller. Generating test cases for specification mining. In *International Symposium on Software Testing and Analysis (ISSTA)*, pages 85–96. ACM, 2010.

[38] V. Dallmeier, C. Lindig, A. Wasylkowski, and A. Zeller. Mining object behavior with ADABU. In *Workshop on Dynamic Systems Analysis (WODA)*, pages 17–24. ACM, 2006.

[39] V. Dallmeier, C. Lindig, and A. Zeller. Lightweight defect localization for Java. In *European Conference on Object-Oriented Programming (ECOOP)*, pages 528–550, 2005.

[40] B. Daniel, D. Dig, K. Garcia, and D. Marinov. Automated testing of refactoring engines. In *European Software Engineering Conference and International Symposium on Foundations of Software Engineering (ESEC/FSE)*, pages 185–194. ACM, 2007.

[41] K. K. Dhara and G. T. Leavens. Forcing behavioral subtyping through specification inheritance. In *International Conference on Software Engineering (ICSE)*, pages 258–267, 1996.

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

[43] T. Elmas, S. Tasiran, and S. Qadeer. VYRD: verifying concurrent programs by runtime refinement-violation detection. In *Conference on Programming Language Design and Implementation (PLDI)*, pages 27–37. ACM, 2005.

[44] D. Engler, D. Y. Chen, S. Hallem, A. Chou, and B. Chelf. Bugs as deviant behavior: A general approach to inferring errors in systems code. In *Symposium on Operating Systems Principles (SOSP)*, pages 57–72. ACM, 2001.

[45] M. D. Ernst. *Dynamically discovering likely program invariants*. PhD thesis, University of Washington, 2000.

[46] M. D. Ernst, J. Cockrell, W. G. Griswold, and D. Notkin. Dynamically discovering likely program invariants to support program evolution. *IEEE Transactions on Software Engineering*, 27(2):213–224, 2001.

[47] M. Erwig and M. M. Burnett. Adding apples and oranges. In *Symposium on Practical Aspects of Declarative Languages (PADL)*, pages 173–191. Springer, 2002.

[48] C. Fetzer and M. Süßkraut. Switchblade: Enforcing dynamic personalized system call models. In *European Conference on Computer Systems (EuroSys)*, pages 273–286. ACM, 2008.

[49] C. Flanagan and S. N. Freund. Atomizer: a dynamic atomicity checker for multithreaded programs. In *Symposium on Principles of Programming Languages (POPL)*, pages 256–267. ACM, 2004.

[50] C. Flanagan and S. N. Freund. FastTrack: efficient and precise dynamic race detection. In *Conference on Programming Language Design and Implementation (PLDI)*, pages 121–133. ACM, 2009.

[51] C. Flanagan, S. N. Freund, and J. Yi. Velodrome: a sound and complete dynamic atomicity checker for multithreaded programs. In *Conference on Programming Language Design and Implementation (PLDI)*, pages 293–303. ACM, 2008.

[52] P. Fonseca, C. Li, and R. Rodrigues. Finding complex concurrency bugs in large multi-threaded applications. In *European Conference on Computer Systems (EuroSys)*, pages 215–228. ACM, 2011.

[53] G. Fraser and A. Zeller. Mutation-driven generation of unit tests and oracles. In *International Symposium on Software Testing and Analysis (ISSTA)*, pages 147–158. ACM, 2010.

[54] M. Gabel and Z. Su. Javert: Fully automatic mining of general temporal properties from dynamic traces. In *Symposium on Foundations of Software Engineering (FSE)*, pages 339–349. ACM, 2008.

[55] M. Gabel and Z. Su. Online inference and enforcement of temporal properties. In *International Conference on Software Engineering (ICSE)*, pages 15–24, 2010.

[56] Q. Gao, W. Zhang, Z. Chen, M. Zheng, and F. Qin. 2ndstrike: toward manifesting hidden concurrency typestate bugs. In *Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 239–250, 2011.

[57] C. Ghezzi, M. Jazayeri, and D. Mandrioli. *Fundamentals of Software Engineering (2nd edition).* Prentice Hall, 2002.

[58] C. Ghezzi, A. Mocci, and M. Monga. Synthesizing intensional behavior models by graph transformation. In *International Conference on Software Engineering (ICSE)*, pages 430–440. IEEE, 2009.

[59] P. Godefroid, N. Klarlund, and K. Sen. DART: directed automated random testing. In *Conference on Programming Language Design and Implementation (PLDI)*, pages 213–223. ACM, 2005.

[60] P. Godefroid and N. Nagappan. Concurrency at microsoft - an exploratory survey. In *Workshop on Exploiting Concurrency Efficiently and Correctly (EC2)*, 2008.

[61] B. Goetz, T. Peierls, J. Bloch, J. Bowbeer, D. Holmes, and D. Lea. *Java Concurrency in Practice*. Addison-Wesley, 2006.

[62] J. Gosling, B. Joy, G. Steele, and G. Bracha. *Java Language Specification, 3rd Edition*. Prentice Hall, 2005.

[63] D. Greenfieldboyce and J. S. Foster. Type qualifier inference for Java. In *Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 321–336. ACM, 2007.

[64] F. Gross, G. Fraser, and A. Zeller. Search-based system testing: High coverage, no false alarms. In *International Symposium on Software Testing and Analysis (ISSTA)*, pages 67–77, 2012.

[65] N. Gruska, A. Wasylkowski, and A. Zeller. Learning from 6,000 projects: Lightweight cross-project anomaly detection. In *International Symposium on Software Testing and Analysis (ISSTA)*, pages 119–130. ACM, 2010.

[66] P. J. Guo, J. H. Perkins, S. McCamant, and M. D. Ernst. Dynamic inference of abstract types. In *International Symposium on Software Testing and Analysis (ISSTA)*, pages 255–265. ACM, 2006.

[67] C. Hammer, J. Dolby, M. Vaziri, and F. Tip. Dynamic detection of atomic-set-serializability violations. In *International Conference on Software Engineering (ICSE)*, pages 231–240. ACM, 2008.

[68] S. Hangal and M. S. Lam. Tracking down software bugs using automatic anomaly detection. In *International Conference on Software Engineering (ICSE)*, pages 291–301. ACM, 2002.

[69] S. Hangal and M. S. Lam. Automatic dimension inference and checking for object-oriented programs. In *International Conference on Software Engineering (ICSE)*, pages 155–165. IEEE, 2009.

[70] J. Henkel, C. Reichenbach, and A. Diwan. Discovering documentation for Java container classes. *IEEE Transactions on Software Engineering*, 33(8):526–543, 2007.

[71] J. L. Henning. SPEC CPU2006 benchmark descriptions. *SIGARCH Computer Architecture News*, 34(4):1–17, 2006.

[72] T. A. Henzinger, R. Jhala, and R. Majumdar. Permissive interfaces. In *European Software Engineering Conference and Symposium on Foundations of Software Engineering (ESEC/FSE)*, pages 31–40. ACM, 2005.

[73] M. Herlihy and J. M. Wing. Linearizability: A correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems*, 12(3):463–492, 1990.

[74] E. W. Høst and B. M. Østvold. Debugging method names. In *European Conference on Object-Oriented Programming (ECOOP)*, pages 294–317. Springer, 2009.

[75] D. Hovemeyer and W. Pugh. Finding bugs is easy. In *Companion to the Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 132–136. ACM, 2004.

[76] V. Jagannath, M. Gligoric, D. Jin, Q. Luo, G. Rosu, and D. Marinov. Improved multithreaded unit testing. In *European Software Engineering Conference and Symposium on Foundations of Software Engineering (ESEC/FSE)*, pages 223–233, 2011.

[77] C. Jaspan and J. Aldrich. Are object protocols burdensome? An empirical study of developer forums. In *Workshop on Evaluation and usability of programming languages and tools (PLATEAU)*, pages 51–56. ACM, 2011.

[78] H. Jaygarl, K.-S. Lu, and C. K. Chang. GenRed: A tool for generating and reducing object-oriented test cases. In *Computer Software and Applications Conference (COMPSAC)*, pages 127–136. IEEE, 2010.

[79] G. Jin, A. V. Thakur, B. Liblit, and S. Lu. Instrumentation and sampling strategies for cooperative concurrency bug isolation. In *Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 241–255. ACM, 2010.

[80] W. Jin, A. Orso, and T. Xie. Automated behavioral regression testing. In *International Conference on Software Testing, Verification and Validation (ICST)*, pages 137–146. IEEE, 2010.

[81] S. C. Johnson. Lint, a C program checker, 1978.

[82] P. Joshi, M. Naik, K. Sen, and D. Gay. An effective dynamic analysis for detecting generalized deadlocks. In *Symposium on Foundations of Software Engineering (FSE)*, pages 327–336. ACM, 2010.

[83] P. Joshi, C.-S. Park, K. Sen, and M. Naik. A randomized dynamic program analysis technique for detecting real deadlocks. In *Conference on Programming Language Design and Implementation (PLDI)*, pages 110–120. ACM, 2009.

[84] S. Joshi, S. K. Lahiri, and A. Lal. Underspecified harnesses and interleaved bugs. In *Principles of Programming Languages (POPL)*, pages 19–30, 2012.

[85] K. Kim, T. Yavuz-Kahveci, and B. A. Sanders. Precise data race detection in a relaxed memory model using heuristic-based model checking. In *Conference on Automated Software Engineering (ASE)*, pages 495–499. IEEE, 2009.

[86] Z. Lai, S.-C. Cheung, and W. K. Chan. Detecting atomic-set serializability violations in multithreaded programs through active randomized testing. In *International Conference on Software Engineering (ICSE)*, pages 235–244. ACM, 2010.

[87] J. L. Lawall and D. Lo. An automated approach for finding variable-constant pairing bugs. In *International Conference on Automated Software Engineering (ASE)*, pages 103–112. ACM, 2010.

[88] D. Lawrie, H. Feild, and D. Binkley. Extracting meaning from abbreviated identifiers. In *Working Conference on Source Code Analysis and Manipulation (SCAM)*, pages 213–222. IEEE, 2007.

[89] D. Lawrie, C. Morrell, H. Feild, and D. Binkley. What's in a name? A study of identifiers. In *International Conference on Program Comprehension (ICPC)*, pages 3–12. IEEE, 2006.

[90] G. T. Leavens and W. E. Weihl. Reasoning about object-oriented programs that use subtypes. In *Conference on Object-Oriented Programming Systems, Languages, and Applications and European Conference on Object-Oriented Programming (OOPSLA/ECOOP)*, pages 212–223, 1990.

[91] O. Lhoták and L. J. Hendren. Evaluating the benefits of context-sensitive points-to analysis using a BDD-based implementation. *ACM Transactions on Software Engineering and Methodology*, 18(1), 2008.

[92] Z. Li and Y. Zhou. PR-Miner: Automatically extracting implicit programming rules and detecting violations in large software code. In *European Software Engineering Conference and Symposium on Foundations of Software Engineering (ESEC/FSE)*, pages 306–315. ACM, 2005.

[93] J. L. Lions. ARIANE 5 flight 501 failure. Report by the inquiry board. European Space Agency, 1996.

[94] B. Liskov. Data abstraction and hierarchy. In *Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, pages 17–34, 1987.

[95] B. Liskov and J. Wing. A behavioral notion of subtyping. *ACM Transactions on Programming Languages and Systems*, 16(6):1811–1841, 1994.

[96] V. B. Livshits and T. Zimmermann. DynaMine: Finding common error patterns by mining software revision histories. In *European Software Engineering Conference and Symposium on Foundations of Software Engineering (ESEC/FSE)*, pages 296–305. ACM, 2005.

[97] D. Lo and S.-C. Khoo. SMArTIC: Towards building an accurate, robust and scalable specification miner. In *Symposium on Foundations of Software Engineering (FSE)*, pages 265–275. ACM, 2006.

[98] D. Lorenzoli, L. Mariani, and M. Pezzè. Automatic generation of software behavioral models. In *International Conference on Software Engineering (ICSE)*, pages 501–510. ACM, 2008.

[99] S. Lu, S. Park, C. Hu, X. Ma, W. Jiang, Z. Li, R. A. Popa, and Y. Zhou. MUVI: Automatically inferring multi-variable access correlations and detecting related semantic and concurrency bugs. In *Symposium on Operating Systems Principles (SOSP)*, pages 103–116. ACM, 2007.

[100] S. Lu, S. Park, E. Seo, and Y. Zhou. Learning from mistakes: a comprehensive study on real world concurrency bug characteristics. In *Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 329–339. ACM, 2008.

[101] S. Lu, J. Tucek, F. Qin, and Y. Zhou. AVIO: detecting atomicity violations via access interleaving invariants. In *Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 37–48. ACM, 2006.

[102] B. Lucia and L. Ceze. Finding concurrency bugs with context-aware communication graphs. In *Symposium on Microarchitecture (MICRO)*, pages 553–563. ACM, 2009.

[103] L. Mariani, F. Pastore, and M. Pezzè. Dynamic analysis for diagnosing integration faults. *IEEE Transactions on Software Engineering*, 37(4):486–508, 2011.

[104] D. Marino, M. Musuvathi, and S. Narayanasamy. LiteRace: effective sampling for lightweight data-race detection. In *Conference on Programming Language Design and Implementation (PLDI)*, pages 134–143. ACM, 2009.

[105] M. C. Martin, V. B. Livshits, and M. S. Lam. Finding application errors and security flaws using PQL: A program query language. In *Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 365–383. ACM, 2005.

[106] S. McCamant and M. D. Ernst. Predicting problems caused by component upgrades. In *European Software Engineering Conference and Symposium on Foundations of Software Engineering (ESEC/FSE)*, pages 287–296. ACM, 2003.

[107] S. McConnell. *Code Complete: A Practical Handbook of Software Construction, Second Edition*. Microsoft Press, 2004.

[108] W. M. McKeeman. Differential testing for software. *Digital Technical Journal*, 10(1):100–107, 1998.

[109] M. Monperrus, M. Bruch, and M. Mezini. Detecting missing method calls in object-oriented software. In *European Conference on Object-Oriented Programming (ECOOP)*, pages 2–25. Springer, 2010.

[110] M. Musuvathi, S. Qadeer, T. Ball, G. Basler, P. A. Nainar, and I. Neamtiu. Finding and reproducing Heisenbugs in concurrent programs. In *Symposium on Operating Systems Design and Implementation*, pages 267–280. USENIX, 2008.

[111] M. Naik, C.-S. Park, K. Sen, and D. Gay. Effective static deadlock detection. In *International Conference on Software Engineering (ICSE)*, pages 386–396. IEEE, 2009.

[112] T. T. Nguyen, H. A. Nguyen, N. H. Pham, J. M. Al-Kofahi, and T. N. Nguyen. Graph-based mining of multiple object usage patterns. In *European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*, pages 383–392. ACM, 2009.

[113] A. Nistor, Q. Luo, M. Pradel, T. R. Gross, and D. Marinov. Ballerina: Automatic generation and clustering of efficient random unit tests for multithreaded code. In *International Conference on Software Engineering (ICSE)*, pages 727–737, 2012.

[114] R. O'Callahan and J.-D. Choi. Hybrid dynamic data race detection. In *Symposium on Principles and Practice of Parallel Programming (PPOPP)*, pages 167–178. ACM, 2003.

[115] J. Offutt, R. T. Alexander, Y. Wu, Q. Xiao, and C. Hutchinson. A fault model for subtype inheritance and polymorphism. In *International Symposium on Software Reliability Engineering (ISSRE)*, pages 84–95. IEEE, 2001.

[116] C. Pacheco, S. K. Lahiri, and T. Ball. Finding errors in .NET with feedback-directed random testing. In *International Symposium on Software Testing and Analysis (ISSTA)*, pages 87–96. ACM, 2008.

[117] C. Pacheco, S. K. Lahiri, M. D. Ernst, and T. Ball. Feedback-directed random test generation. In *International Conference on Software Engineering (ICSE)*, pages 75–84. IEEE, 2007.

[118] N. Palix, G. T. 0001, S. Saha, C. Calvès, J. L. Lawall, and G. Muller. Faults in linux: ten years later. In *Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 305–318. ACM, 2011.

[119] K. Pan, S. Kim, and E. J. W. Jr. Toward an understanding of bug fix patterns. *Empirical Software Engineering*, 14(3):286–315, 2009.

[120] C.-S. Park and K. Sen. Randomized active atomicity violation detection in concurrent programs. In *Symposium on Foundations of Software Engineering (FSE)*, pages 135–145. ACM, 2008.

[121] S. Park, S. Lu, and Y. Zhou. CTrigger: exposing atomicity violation bugs from their hiding places. In *Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 25–36. ACM, 2009.

[122] H. Patil, C. Pereira, M. Stallcup, G. Lueck, and J. Cownie. PinPlay: a framework for deterministic replay and reproducible analysis of parallel programs. In *Symposium on Code Generation and Optimization (CGO)*, pages 2–11. ACM, 2010.

[123] K. Poulsen. Software bug contributed to blackout. SecurityFocus, February 2004.

[124] M. Pradel, P. Bichsel, and T. R. Gross. A framework for the evaluation of specification miners based on finite state machines. In *International Conference on Software Maintenance (ICSM)*, pages 1–10. IEEE, 2010.

[125] M. Pradel and T. R. Gross. Automatic generation of object usage specifications from large method traces. In *International Conference on Automated Software Engineering (ASE)*, pages 371–382. IEEE, 2009.

[126] M. Pradel and T. R. Gross. Detecting anomalies in the order of equally-typed method arguments. In *International Symposium on Software Testing and Analysis (ISSTA)*, pages 232–242. ACM, 2011.

[127] M. Pradel and T. R. Gross. Fully automatic and precise detection of thread safety violations. In *Conference on Programming Language Design and Implementation (PLDI)*, pages 521–530, 2012.

[128] M. Pradel and T. R. Gross. Leveraging test generation and specification mining for automated bug detection without false positives. In *International Conference on Software Engineering (ICSE)*, pages 288–298, 2012.

[129] M. Pradel and T. R. Gross. Automatic testing of sequential and concurrent substitutability. In *International Conference on Software Engineering (ICSE)*, 2013.

[130] M. Pradel, S. Heiniger, and T. R. Gross. Static detection of brittle parameter typing. In *International Symposium on Software Testing and Analysis (ISSTA)*, pages 265–275, 2012.

[131] M. Pradel, C. Jaspan, J. Aldrich, and T. R. Gross. Statically checking API protocol conformance with mined multi-object specifications. In *International Conference on Software Engineering (ICSE)*, pages 925–935, 2012.

[132] W. Pugh and N. Ayewah. Unit testing concurrent software. In *Conference on Automated Software Engineering (ASE)*, pages 513–516. ACM, 2007.

[133] M. K. Ramanathan, A. Grama, and S. Jagannathan. Path-sensitive inference of function precedence protocols. In *International Conference on Software Engineering (ICSE)*, pages 240–250. IEEE, 2007.

[134] S. P. Reiss and M. Renieris. Encoding program executions. In *International Conference on Software Engineering (ICSE)*, pages 221–230. IEEE, 2001.

[135] M. Renieris and S. P. Reiss. Fault localization with nearest neighbor queries. In *International Conference on Automated Software Engineering (ASE)*, pages 30–39. IEEE, 2003.

[136] M. P. Robillard. What makes APIs hard to learn? Answers from developers. *IEEE Software*, 26(6):27–34, 2009.

[137] C. Ruby and G. T. Leavens. Safely creating correct subclasses without seeing superclass code. In *Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, pages 208–228, 2000.

[138] M. Salah, T. Denton, S. Mancoridis, A. Shokoufandeh, and F. I. Vokolos. Scenariographer: A tool for reverse engineering class usage scenarios from method invocation sequences. In *Conference on Software Maintenance (ICSM)*, pages 155–164. IEEE, 2005.

[139] S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, and T. E. Anderson. Eraser: A dynamic data race detector for multithreaded programs. *ACM Transactions on Computer Systems*, 15(4):391–411, 1997.

[140] K. Sen. Effective random testing of concurrent programs. In *Conference on Automated Software Engineering (ASE)*, pages 323–332. ACM, 2007.

[141] K. Sen. Race directed random testing of concurrent programs. In *Conference on Programming Language Design and Implementation (PLDI)*, pages 11–21. ACM, 2008.

[142] O. Shacham, N. Bronson, A. Aiken, M. Sagiv, M. Vechev, and E. Yahav. Testing atomicity of composed concurrent operations. In *Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, pages 51–64, 2011.

[143] Y. Shi, S. Park, Z. Yin, S. Lu, Y. Zhou, W. Chen, and W. Zheng. Do I use the wrong definition? DefUse: Definition-use invariants for detecting concurrency and sequential bugs. In *Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 160–174. ACM, 2010.

[144] Y. Smaragdakis and C. Csallner. Combining static and dynamic reasoning for bug detection. In *International Conference on Tests and Proofs (TAP)*, pages 1–16. Springer, 2007.

[145] A. Snyder. Encapsulation and inheritance in object-oriented programming languages. In *Conference on Object-oriented Programming Systems, Languages, and Applications (OOPSLA)*, pages 38–45, 1986.

[146] F. Sorrentino, A. Farzan, and P. Madhusudan. PENELOPE: weaving threads to expose atomicity violations. In *Symposium on Foundations of Software Engineering (FSE)*, pages 37–46. ACM, 2010.

[147] V. Srivastava, M. D. Bond, K. S. McKinley, and V. Shmatikov. A security policy oracle: detecting security holes using multiple API implementations. In *Conference on Programming Language Design and Implementation (PLDI)*, pages 343–354. ACM, 2011.

[148] C. Szyperski. *Component Software: Beyond Object-Oriented Programming*. Addison-Wesley, 2002.

[149] A. Taivalsaari. On the notion of inheritance. *ACM Computing Surveys*, 28(3):438–479, 1996.

[150] E. Tempero, C. Anslow, J. Dietrich, T. Han, J. Li, M. Lumpe, H. Melton, and J. Noble. Qualitas Corpus: A curated collection of Java code for empirical studies. In *Asia Pacific Software Engineering Conference (APSEC)*, 2010.

[151] E. D. Tempero, J. Noble, and H. Melton. How do Java programs use inheritance? An empirical study of inheritance in Java software. In *European Conference on Object-Oriented Programming (ECOOP)*, pages 667–691. Springer, 2008.

[152] S. Thummalapenta, J. de Halleux, N. Tillmann, and S. Wadsworth. DyGen: Automatic generation of high-coverage tests via mining gigabytes of dynamic traces. In *Conference on Tests and Proofs (TAP)*, pages 77–93. Springer, 2010.

[153] S. Thummalapenta and T. Xie. Alattin: Mining alternative patterns for detecting neglected conditions. In *International Conference on Automated Software Engineering (ASE)*, pages 283–294. IEEE, 2009.

[154] S. Thummalapenta and T. Xie. Mining exception-handling rules as sequence association rules. In *International Conference on Software Engineering (ICSE)*, pages 496–506. IEEE, 2009.

[155] S. Thummalapenta, T. Xie, N. Tillmann, J. de Halleux, and W. Schulte. MSeqGen: Object-oriented unit-test generation via mining source code. In *European Software Engineering Conference and International Symposium on Foundations of Software Engineering (ESEC/FSE)*, pages 193–202. ACM, 2009.

[156] N. Tillmann and W. Schulte. Parameterized unit tests. In *European Software Engineering Conference and Symposium on Foundations of Software Engineering (ESEC/FSE)*, pages 253–262. ACM, 2005.

[157] R. Vallée-Rai, P. Co, E. Gagnon, L. J. Hendren, P. Lam, and V. Sundaresan. Soot - a Java bytecode optimization framework. In *Conference of the Centre for Advanced Studies on Collaborative Research (CASCON)*, pages 125–135. IBM, 1999.

[158] W. Visser, K. Havelund, G. P. Brat, S. Park, and F. Lerda. Model checking programs. *Automated Software Engineering*, 10(2):203–232, 2003.

[159] W. Visser, C. S. Pasareanu, and S. Khurshid. Test input generation with Java PathFinder. In *International Symposium on Software Testing and Analysis (ISSTA)*, pages 97–107. ACM, 2004.

[160] C. von Praun and T. R. Gross. Object race detection. In *Conference on Object Oriented Programming, Systems, Languages and Applications (OOPSLA)*, pages 70–82, 2001.

[161] A. Wasylkowski and A. Zeller. Mining temporal specifications from object usage. In *International Conference on Automated Software Engineering (ASE)*, pages 295–306. IEEE, 2009.

[162] A. Wasylkowski, A. Zeller, and C. Lindig. Detecting object usage anomalies. In *European Software Engineering Conference and Symposium on Foundations of Software Engineering (ESEC/FSE)*, pages 35–44. ACM, 2007.

[163] Y. Wei, Y. Pei, C. A. Furia, L. S. Silva, S. Buchholz, B. Meyer, and A. Zeller. Automated fixing of programs with contracts. In *International Symposium on Software Testing and Analysis (ISSTA)*, pages 61–72. ACM, 2010.

[164] W. Weimer and G. C. Necula. Mining temporal specifications for error detection. In *Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, pages 461–476. Springer, 2005.

[165] W. Weimer, T. Nguyen, C. Le Goues, and S. Forrest. Automatically finding patches using genetic programming. In *International Conference on Software Engineering (ICSE)*, pages 363–374, 2009.

[166] J. Whaley, M. C. Martin, and M. S. Lam. Automatic extraction of object-oriented component interfaces. In *Symposium on Software Testing and Analysis (ISSTA)*, pages 218–228. ACM, 2002.

[167] J. M. Wing and C. Gong. Testing and verifying concurrent objects. *Journal of Parallel and Distributed Computing*, 17(1-2):164–182, 1993.

[168] J. Winther and M. I. Schwartzbach. Related types. In *European Conference on Object-Oriented Programming (ECOOP)*, pages 434–458. Springer, 2011.

[169] T. Xie, D. Marinov, W. Schulte, and D. Notkin. Symstra: A framework for generating object-oriented unit tests using symbolic execution. In *Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, pages 365–381. Springer, 2005.

[170] T. Xie and D. Notkin. Mutually enhancing test generation and specification inference. In *Formal Approaches to Software Testing*, pages 60–69. Springer, 2003.

[171] T. Xie and D. Notkin. Automatic extraction of object-oriented observer abstractions from unit-test executions. In *International Conference on Formal Engineering Methods (ICFEM)*, pages 290–305. Springer, 2004.

[172] J. Yang, D. Evans, D. Bhardwaj, T. Bhat, and M. Das. Perracotta: Mining temporal API rules from imperfect traces. In *International Conference on Software Engineering (ICSE)*, pages 282–291. ACM, 2006.

[173] S. Zhang, D. Saff, Y. Bu, and M. D. Ernst. Combined static and dynamic automated test generation. In *International Symposium on Software Testing and Analysis (ISSTA)*, pages 353–363, 2011.

[174] W. Zhang, J. Lim, R. Olichandran, J. Scherpelz, G. Jin, S. Lu, and T. W. Reps. ConSeq: detecting concurrency bugs through sequential errors. In *Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 251–264, 2011.

[175] W. Zhang, C. Sun, and S. Lu. ConMem: detecting severe concurrency bugs through an effect-oriented approach. In *Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 179–192. ACM, 2010.

[176] W. Zheng, Q. Zhang, M. R. Lyu, and T. Xie. Random unit-test generation with MUT-aware sequence recommendation. In *Conference on Automated Software Engineering (ASE)*, pages 293–296. ACM, 2010.

[177] M. Zhivich and R. K. Cunningham. The real cost of software errors. *IEEE Security & Privacy*, 7(2):87–90, 2009.

[178] H. Zhong, T. Xie, L. Zhang, J. Pei, and H. Mei. MAPO: Mining and recommending API usage patterns. In *European Conference on Object-Oriented Programming (ECOOP)*, pages 318–343, 2009.