Toward Effective Debugging by Capturing and Reusing Knowledge

By

ZHONGXIAN GU
B.S. (Shanghai Jiaotong University) 2008
M.S. (University of California, Davis) 2010

DISSERTATION

Submitted in partial satisfaction of the requirements for the degree of

DOCTOR OF PHILOSOPHY

in

COMPUTER SCIENCE

in the

OFFICE OF GRADUATE STUDIES

of the

UNIVERSITY OF CALIFORNIA

DAVIS

Approved:

Professor Zhendong Su (Chair)

Professor Premkumar T. Devanbu

Professor Todd J. Green

Committee in Charge
2013

Contents

Zhongxian Gu
December 2013
Computer Science

Toward Effective Debugging by Capturing and Reusing Knowledge

Abstract

Debugging is an arduous task — localizing and resolving bugs pervade the software development process. The cost of software maintenance accounts for two-thirds of the overall software production cost. Correctly and quickly fixing a bug reduces cost and improves software quality. This dissertation presents three complementary research efforts that target and ease different stages of debugging.

The first concerns the testing and verification of bug fixes. It introduces and formalizes the *bad fix* problem: a fix is bad if it does not properly fix the bug or introduces new bugs. Bad fixes are difficult to detect as they successfully pass the original failing test cases and are more costly to fix. Our research proposes novel approaches to detect bad fixes to avoid them in an early stage. The bad fix problem emphasizes the importance and difficulty of debugging and helps understand and ease the debugging process.

The second aims to help developers fix bugs. Our key insight is to leverage existing debugging knowledge to fix new bugs. We hypothesize that with respect to the millions of already encountered bugs, unique and new bugs are rare — most bugs have similar bugs. We have designed and developed OSCILLOSCOPE, a bug repository and search engine to allow developers search for similar bugs. Learning from similar bugs and how they were fixed, developers can more quickly understand and propose correct fixes to bugs.

To ease the use of OSCILLOSCOPE and understand how developers debug, we introduce IDEPP, an infrastructure that systematically captures fine-grained IDE interactions. IDEPP forms the third part of this dissertation. It can serve as the basis to support an ecosystem of user-aware applications including OSCILLOSCOPE and others. In particular, IDEPP not only facilitates the use of OSCILLOSCOPE, but also enables other general applications to improve programming productivity.

Acknowledgments and Thanks

Many people have provided invaluable support to me during my graduate studies in University of California, Davis.

First, I want to express my deeply-felt thanks to my PhD advisor, Professor Zhendong Su, for supporting me during the past five years. There is an old saying spread among the prospective PhD students, "The most important factor that drives your PhD career and near future is not the ranking of your university, not the reputation of your research lab, but the personality of your advisor". I cannot agree more to this old saying after my PhD career. I feel so lucky that I can have Zhendong, such a brilliant, patient, and supportive person to be my doctoral advisor. Zhendong is someone you will instantly love and never forget once you meet him. I hope that I could be as lively, enthusiastic, and energetic as Zhendong towards research. He spent immeasurable time and efforts training me to be a researcher. He provided me with ideas and yet allowed significant freedom to let me pursue my research. He trained me in writing and presenting research. He sets high standards for all of our work: as he always said, "If you do not feel proud of your own research, then do not do it". I greatly appreciate everything he has done for me.

I would not have contemplated this road if not for my parents, Ping Zhou and Qiqi Gu. They gave me birth, love and pave me the road for pursuing my dream. Especially, I want to express my great-thankfulness to my mother Ping who played a very important role during all my life. She helped me to establish perseverance, patience, and all other personalities that I need to be a good researcher and person.

I thank my colleagues and collaborators who have provided invaluable supports during my graduate study. I thank Lingxiao for helping me leasing my apartment before I went to USA. I thank Sophia Sun and Dennis Xu, who offered me the first lunch and helped me to open my first bank account when I was locked out from my apartment when I just landed in USA. I thank Earl

Barr, who acted as my second advisor during my PhD career. Earl helped me through all the aspects that required to be a researcher in a different country: English as a secondary language, technical writing, research altitude. Most importantly, his enthusiasm towards research is and will always be a great model for me to pursue. I thank the following colleagues that frequently gave very useful feedback on research ideas, paper drafts, and practice talks to help me improve the quality of my work: Mehrdad Afshari, Chris Bird, Mark Gabel, Taeho Kwon, Vu Minh Le, Andreas Saebjoernsen, Drew Schleck, Sophia Sun, Thanh Vo, and Dennis Xu.

I also want to express my heartfelt thankfulness to my girl friend, Siqi Fan, who brought love , happiness and unconditional support to me. Her accompanies really helps me go through all the hard times during my PhD career and gradually makes me a man that is ready to hold a family. I thank all my friends in Davis that gave me a wonderful and unforgettable five-year memory there.

Chapter 1

Introduction

As software systems become increasingly sophisticated, complex, and ubiquitous, it is inevitable that they are often shipped with bugs. During the course of a project, its development team may continuously encounter and receive a large number of bugs over a long period of time. For example, a total of 4,414 bugs were reported for the Eclipse project in 2009. Localizing and fixing bugs pervade the software development process.

Although much research has devoted to reduce the burden of debugging, it is still an arduous task to debug software. According to IDC [43], software maintenance cost $86 billion in 2005, accounting for as much as two-thirds of the overall cost of software production. During software maintenance, programmers typically spend 50–80% of their time on debugging. Brian Kernighan even commented: "Debugging is twice as hard as writing the code in the first place. Therefore, if you write the code as cleverly as possible, you are, by definition, not smart enough to debug it."

Fixing a bug correctly the first time it is encountered will reduce cost and improve software quality. The central theme of this dissertation is to help programmers achieve this goal. It is organized into three chapters, each of which forms a distinct contribution. Chapter 2 introduces and formalizes the *bad fix problem*. Informally, a fix is bad if it does not properly resolve the bug or introduces new bugs. This chapter also proposes a novel approach to detect and avoid a bad fix in the

early stage of development. Chapter 3 proposes novel tool and language support to help programmers identify similar bugs and reuse debugging knowledge to fix new bugs. Chapter 4 motivates and introduces a new IDE infrastructure for capturing systematic, fine-grained user interactions. The infrastructure helps study how users debug in the IDE and enables new applications to aid debugging and beyond.

Chapter 2: Fixation Programmers spend much effort developing fixes, however, they may not pay close enough attention to verify the fixes. When testing a fix, a programmer usually re-run a program with the original bug-triggering inputs. If no errors occur, the developer usually deems the bug has been fixed and shifts to the next task. Folklore suggests that bug fixes are frequently buggy, either failing to handle all bug-triggering inputs or introducing new bugs. Our research, introduces and formalizes the bad fix problem. In particular, we formalize two criteria to determine whether a fix is bad: *coverage* and *disruption*. The coverage of a fix measures the extent to which the fix correctly handles all inputs that may trigger a bug, while disruption measures the deviation from the program's intended behavior after the application of a fix. To compute the coverage and disruption of a fix, we also introduce the novel notion of distance-bounded weakest precondition as the basis for developing a practical tool. To validate our approach, we have implemented FIXATION, a prototype that automatically detects bad fixes for Java programs. Our evaluation shows that FIXATION is able to successfully detect extracted bad fixes from real-world software.

Chapter 3: Oscilloscope Over the many decades of software development, a large number of bugs have been encountered. Some, among the millions that have occurred, are similar to some other existing bugs. A fix for a similar bug may help a developer understand a new bug, or even directly fix it. Studying bugs with similar symptoms, programmers may determine how to detect or resolve them. In this research, we advocate the systematic capture and reuse of debugging

knowledge — most of which is currently wasted — to speed up debugging. We introduce novel tool and language support to help programmers accurately and efficiently identify similar bugs. To this end, we have developed OSCILLOSCOPE, an Eclipse plugin, for finding similar bugs, and its supporting infrastructure. At the heart of this infrastructure is our bug query language (BQL), a flexible query language that can express a wide variety of queries over traces. We demonstrate the utility and practicality of our approach via the collection and study of bugs from the Apache Commons and Mozilla Rhino projects.

Chapter 4: IDEPP Although OSCILLOSCOPE provides an Eclipse plugin to integrate more easily into the development process, it still requires much effort from developers to utilize it, in particular, writing meaningful and useful queries. In this research, we intend to automate this task by studying how programmers debug, extract relevant domain knowledge and use it to guide OSCILLOSCOPE to search for similar bugs. To this end, we have developed IDEPP, an Eclipse plugin that comprehensively and systematically captures fine-grained IDE interactions. These interactions can reflect a developer's thought process and work habits. By capturing and exploiting comprehensive, fine-grained interactions, OSCILLOSCOPE can leverage the captured relevant debugging contexts to help compose queries. We also envision the establishment of an ecosystem of IDEPP-based applications and extensions to exploit this personalized awareness. Examples include the dynamic personalization of IDE interfaces, training users how to use IDEs more effectively, helping users follow best practices, and alerting users of relevant features for a user's current task. We have developed four proof-of-concept IDEPP applications to illustrate the promise of an ecosystem of user-aware applications and the need for capturing comprehensive, fine-grained interactions.

Chapter 2

FIXATION: Has the Bug Really Been Fixed?

Software has bugs, and fixing those bugs pervades the software engineering process. It is folklore that bug fixes are often buggy themselves, resulting in *bad fixes*, either failing to fix a bug or creating new bugs. To confirm this folklore, we explored bug databases of the Ant, AspectJ, and Rhino projects, and found that bad fixes comprise as much as 9% of all bugs. Thus, detecting and correcting bad fixes is important for improving the quality and reliability of software. However, no prior work has systematically considered this *bad fix problem*, which this research introduces and formalizes. In particular, the research formalizes two criteria to determine whether a fix resolves a bug: *coverage* and *disruption*. The coverage of a fix measures the extent to which the fix correctly handles all inputs that may trigger a bug, while disruption measures the deviations from the program's intended behavior after the application of a fix. This research also introduces a novel notion of *distance-bounded weakest precondition* as the basis for the developed practical techniques to compute the coverage and disruption of a fix.

To validate our approach, we implemented Fixation, a prototype that automatically detects bad fixes for Java programs. When it detects a bad fix, Fixation returns an input that still triggers the

bug or reports a newly introduced bug. Programmers can then use that bug-triggering input to refine or reformulate their fix. We manually extracted fixes drawn from real-world projects and evaluated Fixation against them: Fixation successfully detected the extracted bad fixes.

2.1 Introduction

According to IDC [43], software maintenance cost $86 billion in 2005, accounting for as much as two-thirds of the overall cost of software production. Developers spend 50–80% of their time looking for, understanding, and fixing bugs [12]. Fixing bugs correctly the first time they are encountered will save money and improve software quality.

Researchers have paid great attention to detecting and classifying bugs to ease developers' work [3, 14, 18, 21, 24, 38, 72, 82, 87]. In contrast, not much effort has been expended on bug fixes. When testing a fix, programmers usually rerun a program with the bug-triggering input. If no error occurs, programmers, hurried and overburdened as they usually are, often move on to their next task, thinking they have fixed the bug. Folklore suggests that bug fixes are frequently bad, either by failing to *cover*, *i.e.* handle, all bug-triggering inputs, or by introducing *disruptions*, *i.e.* new bugs. A bad fix can decrease the quality of a program by concealing the original bug, rendering it more subtle and increasing the cost of its fix. Thus, detecting and correcting bad fixes as soon as possible is important for the quality and reliability of software.

To gain insight into the prevalence and characteristics of bad fixes, we explored the Bugzilla databases [10] of the Ant, AspectJ and Rhino projects under Apache, Eclipse and Mozilla foundations. At the time of our survey, these databases contained entries to July 2009. In Bugzilla, a bug is *reopened* if it fails regression testing or a symptom of the bug recurs in the field. We hypothesized that a reopened bug might indicate a bad fix. We read the comment histories of reopened bugs to judge whether or not the bug was reopened due to a bad fix. Programmers sometimes even admit they committed a bad fix: in our survey, we found statements like "Oh, I am sorry I didn't

consider that possibility" and "Oops, missed one code path." Of all reopened bugs, we found that bad fixes account for 66% in Ant, 73% in AspectJ, and 80% in Rhino. Bugs reopened because they failed a regression test belong to the disruption dimension of a bad fix. In our preliminary findings, reopened bugs comprise 4–7.25% of *all* bugs in these three projects[1]. We also found that 38–50% of bugs reopened due to a bad fix either had duplicates or blocked other bugs and were therefore linked to other bugs in Bugzilla. We found a total of $1,977$ bad fixes from reopened bugs[2], and an additional 830 duplicates (*i.e.* bugs marked as a duplicate of a bad fix), comprising 9% of the total bugs ($31,201$) in the Apache database. Bad fixes need not manifest in reopened bugs; we focused on reopened bugs because they often make bad fixes easier to identify. Our manual study undercounts the prevalence of bad fixes in the studied projects, although we cannot say by how much.

In this research, we describe the first systematic treatment of the bad fix problem: we introduce and formalize the problem, and present novel techniques to help developers assess the quality of a bug fix. We deem a fix bad if it fails to cover all bug-triggering inputs or introduces new bugs. An ideal fix covers all bug-triggering inputs and introduces no new bugs. We define two criteria to determine whether or not a fix resolves a bug — coverage and disruption:

Coverage: Many inputs may trigger a bug. The *coverage* of a fix measures the extent to which the fix correctly handles all bug-triggering inputs.

Disruption: A fix may unexpectedly change the behavior of the original program. *Disruption* counts these deviations from the program's intended behavior introduced by a fix.

Given a buggy program, a bug-triggering input that results in an assertion failure, a test-suite, and a fix, the *bad fix problem* is to determine the coverage and disruption of the fix.

[1]The absolute numbers are $\frac{377}{5200}$ for Ant, $\frac{86}{2162}$ for AspectJ, and $\frac{38}{847}$ for Rhino. The number is $\frac{2939}{31201}$ (9.4%) across all Apache projects.

[2]Here we restricted ourselves to bugs that had been reopened, but not marked as duplicates.

In theory, Dijkstra's weakest precondition (WP) [17] can be used to calculate the coverage of a fix. We start from the manifestation of the bug in the buggy version of the program to discover the set predicate of bug-triggering subset of the program's input domain. Then we would symbolically execute the fixed program to learn whether, starting from that set predicate, inputs still exist that can trigger the bug. However, Dijkstra's WP computation depends on loop invariants and must contend with paths exponential in the number of branches.

We propose *distance-bounded weakest precondition (WP$_d$)* to perform the WP calculation over a set of paths near a concrete path. Given a path and a distance budget, WP$_d$ produces a set of paths and computes the disjunction of the WP of each path. In the context of the bad fix problem, the bug-triggering input induces this concrete path. This process is sound and yields an under-approximation of the set of bug-triggering inputs. We can improve our under-approximation by increasing the distance budget. Indeed, in the limit when the distance budget tends to infinity, WP$_d$ is precisely Dijkstra's WP.

WP$_d$ offers two practical benefits. First, because we operate directly on paths, we avoid both the loop invariant requirement and the path-explosion problem. Second, it is our intuition that paths closer to the bug-triggering path are more likely to be related to the same bug and more error-prone, so adding them is likely to quickly approximate the bug-triggering input domain at low cost. Kim *et al.* showed that when a bug occurs in a file, more bugs are likely to occur in that file, a phenomenon they name temporal locality [49]. Their results can be put another way: defects are lexically clustered, which supports our intuition since many execution paths that are close to each other are also lexically close.

This approach may appear circuitous: why not apply WP$_d$ directly to the fixed program to see if we can find an input that triggers the assertion failure? The problem is that the original buggy input no longer triggers the bug in the fixed program. Thus, the concrete path that triggers the bug in the buggy version of the program no longer reaches the assertion and may not even exist in fixed program. Computing WP based on this false path may lead to spurious or incorrect results.

Regression testing is a measure of our disruption criterion; a project's test suite is a parameter of the bad fix problem to take advantage of this fact. We combine random and regression testing to calculate the disruption of a fix.

To demonstrate the feasibility of our approach, we implemented a prototype, Fixation, which automatically detects bad fixes in Java programs. Given the buggy and fixed, versions of a program, a test-suite, and a bug-triggering input, Fixation solves the bad fix problem. Our tool currently supports Java programs with conditionals in Boolean and integer domains. When it detects a coverage failure, it outputs a counterexample that triggers the bug in the fixed program; when it detects a disruptive fix, it reports the failing test cases or inputs. From examining the counterexample or the failing test cases, programmer can understand why the fix did not work and improve it.

The main contributions of this research are:

- We introduce the bad fix problem and provide empirical evidence of its importance by exploring the bug databases of three real projects to find that bad fixes accounts for as much as 9% of all bugs.

- We formalize the bad fix problem and propose distance-bounded weakest precondition (WP_d), a novel form of weakest precondition, well-suited for the bad fix problem, that restricts the weakest precondition computation to a subset of the paths in a program's control flow graph.

- We implemented a prototype, called Fixation, to check the coverage and disruption of a fix. We evaluated our prototype to demonstrate the feasibility of our approach: Fixation detects bad fixes extracted from real-world programs.

The rest of this chapter is as follows. In Section 2.2, we illustrate the problem with actual bad fixes and show how our technique can detect them. Section 2.3 formalizes our criteria for a bad fix

```
1  // no idea what to do if it's a TAIL_CALL
2  if ( fun instanceof NoSuchMethodShim
3       && op != Icode_TAIL_CALL){
4
5    // get the shim and the actual method
6    NoSuchMethodShim =(NoSuchMethodShim)fun;
7    Callable noSuchMethodMethod =
8        noSuchMethodShim.noSuchMethodMethod;
9    ...
10 }
```

Figure 2.2.1: First fix of `NoSuchMethod`.

and presents the detailed technique to measure them. We describe our prototype implementation and evaluation results in Section 2.4. Finally, we survey related work (Section 2.5) and discuss future work (Section 2.6).

2.2 Illustrative Example

This section describes an actual sequence of bad fixes for a bug from the Rhino project, and how our approach would have helped.

Rhino is an open-source JavaScript interpreter written in Java. JavaScript allows programmers to define `_noSuchMethod_`, a special method that the JavaScript interpreter invokes, instead of raising an exception, when an undefined method is called on an object. The bug, which we name `NoSuchMethod`, was a lack of support for this `_noSuchMethod_` mechanism. Its fix is not complicated; the final patch was less than 100 lines. However, due to bad fixes, the bug was reopened twice and three fixes were committed in three months.

Figure 2.2.1 contains the first committed fix. On Line 1, the programmer admits that he was not sure whether this fix covered all relevant inputs. The fix adds an `if`-block which, when an undefined method has been called, extracts `noSuchMethodMethod` from `NoSuchMethodShim` and dispatches the undefined method on it, passing the original test case. However, the clause "`op`

```
1  if ( fun instanceof NoSuchMethodShim ) {
2  if ( fun instanceof NoSuchMethodShim
3       && op != Icode_TAIL_CALL) {
4
5       // get the shim and the actual method
6       NoSuchMethodShim = (NoSuchMethodShim)fun;
7       Callable noSuchMethodMethod =
8            noSuchMethodShim.noSuchMethodMethod;
9       ...
10      if ( op == Icode_TAIL_CALL ) {
11          callParentFrame = frame.parentFrame;
12          exitFrame(cx, frame, null);
13      }
14      ...
15 }
```

Figure 2.2.2: Second fix of `NoSuchMethod`. Green, normal weight lines indicate changes added in this fix; red, strikeout lines indicate those removed; and gray lines are those left unchanged.

`!= Icode_TAIL_CALL`" could be false for an undefined method call. The programmer missed this case. Under our criteria, this fix fails the coverage check. Given buggy and fixed versions of the program, Fixation would compute the predicate of the bug-triggering input domain and symbolically execute the fixed program with that predicate as the initial precondition. Upon reaching the exception, Fixation would determine the fix to be bad, and return the counterexample "`fun instanceof NoSuchMethodShim` $\wedge$ `op == Icode_TAIL_CALL`" to the programmer. In this example, Fixation can exploit the common idiom of asserting `false` at a code path not expected to be reached; in general, however, the assertion that captures a bug can be more complex.

After the bug was reopened, the programmer refined the fix, as shown in Figure 2.2.2. The clause that restricted the operation mode was dropped. Inside this `if`-block, the programmer added a block to deal with the case when operation mode was set to `Icode_TAIL _CALL`. The fix handles all inputs that triggered the original bug. However, the fix failed when subjected to regression testing. It fails the disruption check of a bad fix: it excised the bug that motivated its application at the cost of introducing new bugs.

Finally, the programmer committed a third version of the fix, which resolved the bug and passed the regression tests. This sequence of fixes shows how easy it is to write a bad fix. Programmers considering only of a subset of the bug-triggering input domain are likely to miss conditions and execution paths. Our technique can help programmers detect these conditions earlier and write better fixes more quickly.

2.3 Approach

To begin, we formalize our problem domain, then define the coverage and disruption of a fix. We abstract the bug b as a failure of the assertion φ. Ideally, we would directly compute whether Dijkstra's WP from φ in the fixed program is false. This computation requires loop invariants and must contend with the path-explosion problem. We have a bug-triggering input and its induced concrete failing path. The key idea of our approach for computing whether a fix covers all inputs that can trigger a bug is to leverage that failing path to compute a sound under-approximation of the bug-triggering input domain, then test the fixed program against inputs from that domain. In Section 2.3.2, we introduce WP_d to compute that sound under-approximation. Section 2.3.3 shows how we use that under-approximation to symbolically execute the fixed program to calculate a counterexample to the fix. We close, in Section 2.3.4, by presenting our algorithm for computing fix disruption which combines regression and random testing.

2.3.1 The Bad Fix Problem

A good bug fix eliminates the bug for all inputs without introducing new bugs. We define two criteria to measure these dimensions — *coverage* and *disruption*.

We model a program $P : I \to O$ as a function in terms of its input/output behavior. We assume that the bug b causes an assertion failure in the buggy program P_b and that we know a bug-triggering input i_b such that Equation 2.3.1 holds. Concretely, i_b represents a failing test case, *i.e.*,

the output $P_b(i_b)$ violates the assertion φ.

$$P_b(i_b) \not\models \varphi \quad \text{(or equivalently, } P_b(i_b) \models \neg\varphi) \tag{2.3.1}$$

Many inputs may trigger $\neg\varphi$. Definition 1 specifies this set.

DEFINITION 1 (Bug-triggering input domain)**.**

$$\tilde{i}_b = \{i \in I : P_b(i) \models \neg\varphi\}$$

A bug fix f creates a new version of the program P_f. At the very least, $P_f(i_b) \models \varphi$, but P_f may not handle all of $\tilde{i}_b$. Definition 2 defines the subset of $\tilde{i}_b$ that P_f handles.

DEFINITION 2 (Covered Bug-Triggering Inputs)**.**

$$\hat{i}_b = \{i \in \tilde{i}_b : P_f(i) \models \varphi\}$$

Ideally, a fix f eliminates the bug b and covers all of $\tilde{i}_b$. With respect to $\tilde{i}_b$, the set $\hat{i}_b$ indicates the degree to which a fix achieves this goal, *viz.* the first dimension of fix quality, its *coverage*. We use $cov(f)$ to denote the coverage of a fix f.

By definition, P_f is correct, relative to b, for the inputs in $\hat{i}_b$. Outside of $\tilde{i}_b$, a bad fix f may introduce new bugs. Definition 3 captures these bugs.

DEFINITION 3 (Introduced Bugs)**.** *Let P^o be the correct oracle for P,* i.e.*, for all inputs, P^o produces the desired output.*

$$B_f = \{i \in I \setminus \tilde{i}_b : P_f(i) \neq P^o(i)\} \tag{2.3.2}$$

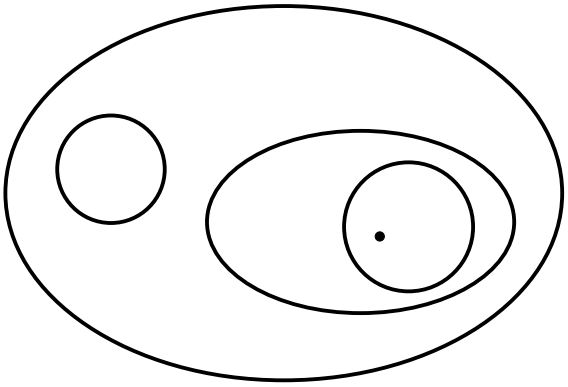


Figure 2.3.1: A program's input domain I, the known input i_b that triggers the bug b, all inputs that trigger the bug $\tilde{i}_b$, those inputs the fix f handles $\hat{i}_b$, new bugs B_f that f may introduce, and their inter-relationships.

The *disruption* of the fix f is the set of new deviating input values it introduces, *viz.* the set B_f. When f introduces no new bugs, $B_f = \emptyset$ and f is not disruptive. Since we do not have a program oracle in general, we approximate P^o with the program's test suite T and P_b, the buggy version of the program. Section 2.3.4 presents the algorithm we use to compute disruption. Along the disruption dimension, we compare the quality of two fixes in terms of the B_f sets they induce.

An *ideal fix* covers all bug-triggering inputs and introduces no disruptions:

$$\hat{i}_b = \tilde{i}_b \ \wedge \ B_f = \emptyset. \tag{2.3.3}$$

Figure 2.3.1 illustrates the interrelations of the sets defined in this section. With our coverage and disruption properties in hand, we now define the bad fix problem.

DEFINITION 4 (Bad Fix Problem)**.** *Given a buggy program P_b, a bug-triggering input i_b, a test suite $T : I \rightarrow O$ (modeled as a* partial function *from I to O), and the fix f, determine the coverage and disruption of f.*

We can also use our criteria to partially order fixes for the same bug. The fix f_a is better than

f_b if and only if

$$cov(f_b) \subseteq cov(f_a) \ \wedge \ B_{f_a} \subseteq B_{f_b}. \tag{2.3.4}$$

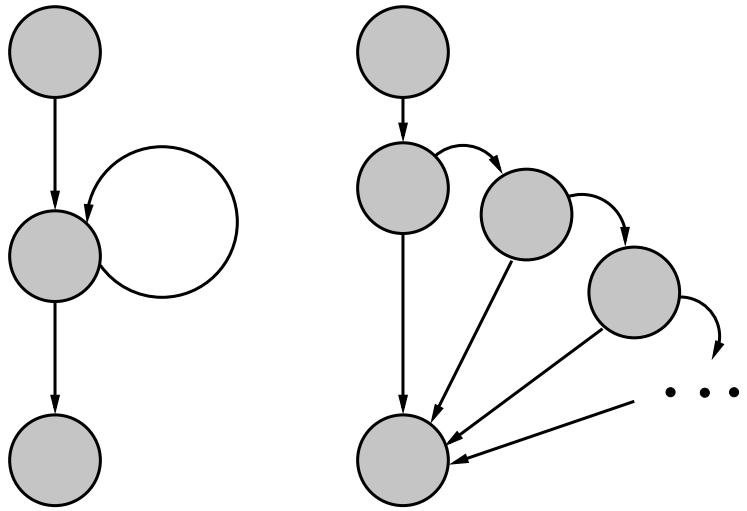
2.3.2 Distance-Bounded Weakest Precondition

To determine whether a fix covers the bug-triggering, we introduce the concept of distance-bounded weakest precondition (WP_d) which generalizes Dijkstra's weakest precondition (WP). WP_d restricts the weakest precondition computation to a subset of the paths near a distinguished path in the interprocedural control flow graph (ICFG) of a program. In the context of the bad fix problem, the distinguished path is Π_{i_b}, the concrete path induced by the known bug-triggering input i_b. In this section, we explain how WP_d traverses the ICFG of a program and uses Levenshtein edit distance [55] to construct the subset of simple paths over which WP_d computes the weakest precondition. By considering only a subset of simple paths, WP_d mitigates the path-explosion problem and does not need loop invariants:

$$\text{WP}_d : \text{Programs} \times \text{Predicates} \times \text{Paths} \times \mathbb{N}_0 \rightarrow \text{Predicates}. \tag{2.3.5}$$

Equation 2.3.5 defines the signature of WP_d, which adds a path Π and a distance d to the signature of standard WP. An application of $\text{WP}_d(P, \varphi, \Pi, d)$ first generates the set C of candidate paths at most d distance from Π, then computes the standard weakest precondition over only the candidate paths in C. Equation 2.3.6 defines the candidate paths WP_d considers. The metric Δ computes the distance of two paths. Currently, we assign a symbol to every edge in the program's ICFG, map every path to a string, and use Levenshtein distance as our metric Δ.

$$C = \{s \in \text{Paths} : \Delta(s, \Pi) \leq d\} \tag{2.3.6}$$



(a) CFG with loop (b) ICFG$_\infty$

Figure 2.3.2: A CFG and its equivalent ICFG$_\infty$.

An ICFG represents loops with backedges. Thus, a concrete path that iterates in a loop is not a simple path in an ICFG. To statically extract paths and compute their distance, we eliminate all backedges by infinitely unrolling all loops to form an infinitely unrolled ICFG, denoted ICFG$_\infty$. Figure 2.3.2 shows a loop in an ICFG and its unrolled representation in an ICFG$_\infty$. All executions have simple paths in an ICFG$_\infty$. ICFG$_n$, a finite subgraph of an ICFG$_\infty$, unrolls all loops n times. All terminating programs iterate each loop a finite number of times, and can be captured by an ICFG$_n$. In particular, all paths within distance d of Π_{i_b} statically exist in an ICFG$_n$ for some n.

$$\eta_G : V \to 2^V \tag{2.3.7}$$

$$l : \Sigma^* \times \Sigma^* \to \mathbb{N}_0 \tag{2.3.8}$$

The function η_G in Equation 2.3.7 returns the neighbors of a vertex in the graph G, and l in Equation 2.3.8 calculates the Levenshtein distance of two strings. Algorithm 1 uses these functions to calculate C. For the sequence s, Algorithm 1 uses the notation $s[i]$ to denote the i^{th}

Algorithm 1 Generate paths Levenshtein distance d from Π

Require: d {distance}
Require: ICFG
Require: Π
  G = reverseArcs(ICFG)
  Π^r = reverse(Π)
  result-paths $\leftarrow \emptyset$
  q.enqueue($\langle \Pi^r[0], \langle\rangle \rangle$) {vertex, path}
  while not q.empty? **do**
    $\langle v, p \rangle$ = q.dequeue()
    if $v = \Pi^r[-1] \wedge l(p, \Pi^r) \leq d$ **then**
      result-paths $\leftarrow$ result-paths $\cup \{p\}$
    end if
    $e = \min(|p| - 1, |\Pi|)$
    if $l(p, \Pi^r[0, e]) \leq 2d$ **then**
      p.append(v)
      q.enqueue($\langle n, p \rangle), \forall n \in \eta_G(v)$)
    end if
  end while
  return result-paths

component of s, $s[-1]$ to denote the last component of s, and $s[i, j],$ for $0 < i < j \leq |s|$ to denote the substring from i to j in s.

Algorithm 1 reverses Π and the arcs in the given ICFG, before traversing each path until it exceeds a distance budget of $2d$. Paths that reach the entry are retained only if they pass the more stringent distance d as they can no longer become closer to Π. Algorithm 1 handles either an ICFG or an ICFG$_\infty$, but is easier to understand when traversing an ICFG$_\infty$.

The set of candidate paths C that Algorithm 1 outputs may contain infeasible paths, such as ones that iterate a loop once more than its upper bound. Such paths are discarded in WP$_d$'s second phase. Each path in C is traversed. When a path reaches a node, a theorem-solver attempts to satisfy its current predicate. A path whose predicate is unsatisfiable is discarded. For example, a path that iterates a loop beyond its upper bound generates a predicate similar to $i = 4 \wedge i < 4$. The

weakest preconditions of satisfiable paths are computed and combined to form a disjunction:

$$\mathrm{WP}_d(P, \varphi, \Pi, d) = \bigvee_{c \in C} \mathrm{WP}(c, \varphi). \tag{2.3.9}$$

As d increases, WP_d calculates the weakest precondition of a larger subset of the paths over which standard WP computes; in the limit, WP_d becomes WP:

$$\lim_{d \to \infty} \mathrm{WP}_d(P, \varphi, \Pi, d) = \mathrm{WP}(P, \varphi). \tag{2.3.10}$$

Under standard WP, the number of paths grows exponentially with the number of jumps in the target program. Polymorphism exacerbates this problem in object-oriented programs. In WP_d, the distance factor d controls the number of paths used in the weakest precondition computation. When $d = 0$, WP_d only computes WP on the path Π. Varying d allows one to trade off the precision of the computed predicate against the efficiency of its computation. In contrast with standard WP, moreover, WP_d selects paths close to the erroneous path Π_{i_b}. Thus, WP_d can, in principle, find a counterexample using fewer resources than standard WP.

Unlike WP, WP_d does not need loop invariants. The difficulty of deriving loop invariants has hampered the application of WP. Indeed, current tools have adopted various heuristics, such as iterating a loop 1.5 times (loop condition twice, its body once), to circumvent the lack of loop invariants [11, 21]. Under WP_d, edit distance determines candidate paths and therefore the number of the loop iterations of each loop along the path. WP_d supports context-sensitivity via cloning; the distance budget determines whether WP_d explores a path with one more or one fewer recursive calls. Though edit distance may construct infeasible paths, WP computation along such a path will produce an unsatisfiable predicate which will cause WP_d to discard the path.

2.3.3 Detecting Violations of Coverage

With the help of WP_d, the coverage of a fix f, $cov(f)$, can be computed in three main steps, as shown in Equation 2.3.11:

$$\overbrace{\exists x_1 \cdots x_n \; [\mathrm{SE}(P_f, \underbrace{\mathrm{WP}_d(P_b, \neg\varphi, \overbrace{e(P_b(i_b))}^{\text{step 1}}, d)}_{\text{step 2}} \wedge \varphi)]}^{\text{step 3}} \tag{2.3.11}$$

1. Extract the concrete path Π_{i_b} that the buggy input i_b induces;

2. Compute $\mathrm{WP}_d(P_b, \neg\varphi, \Pi_{i_b}, d) = \alpha$; and

3. Symbolically execute P_f using α to derive P_f's postcondition ψ, and eliminate the non-input variables x_i from the clause $\psi \wedge \varphi$ to yield $cov(f)$.

In the first step, we run $P_b(i_b)$, then apply e to extract Π_{i_b}. Recall that φ is the failing assertion. In the second step, α under-approximates the set predicate of $\tilde{i}_b$, the true bug-triggering input domain. In this step, we compute α for various values of d, depending on resource constraints. The precision of our under-approximation depends on d. Starting from the weakest precondition computed for various d in step 2 anded with the assertion φ, the third step uses symbolic execution [50] to compute the set of bug-triggering inputs handled by the bug fix. We eliminate non-input variables, *i.e.* intermediate and output variables, from the clause, as they are irrelevant to coverage, which concerns only inputs.

To decide whether f is a bad fix in terms of coverage, we check the validity of $\psi \rightarrow \varphi$. If it is valid, the fix does not violate the coverage requirement. Otherwise, we deem f a bad fix and report any counterexamples to the validity of $\psi \rightarrow \varphi$ as new bug-triggering inputs. Our assertion that f does not cover $\tilde{i}_b$ is sound because α under-approximates $\tilde{i}_b$, *i.e.*, $\{i \in I : \alpha\} \subseteq \tilde{i}_b$.

Algorithm 2 Compute the disruption of a fix

Require: I {The program's input domain}
Require: p_b {The buggy version of the program}
Require: p_f {The fixed version of the program}
Require: $T : I \to O$ {The program's test suite}
1: $R = \emptyset$
2: $\forall (i,o) \in T$ **do** {Standard regression testing}
3: **if** $p_f(i) \neq o$ **then**
4: $R \leftarrow R \cup \{i\}$
5: **end if**
6: **end for**
7: $\forall i \in$ a random subset of I **do**
8: **if** $p_b(i) \Rightarrow \wedge p_b(i) \neq p_f(i)$ **then**
9: $R \leftarrow R \cup \{i\}$
10: **end if**
11: **end for**
12: **return** R

2.3.4 Detecting Violations of Disruption

To measure the disruption of the fix f, we first run P_f on the test suite, as is conventional, because test suites are crafted to exercise important execution paths [34, 77]. Each test failure is a disruption. Given a specification of a program's input I, we then randomly choose an input $i \in I \setminus \tilde{i}_b$. We compare the output of P_b to P_f to find errors not anticipated by the test suite. Each time $P_b(i) \neq P_f(i)$, we have found another disruption. Because $\hat{i}_b$ under-approximates the actual bug-triggering input domain $\tilde{i}_b$, we ignore inputs that trigger $\neg\varphi$ in P_b.

Algorithm 2 computes the disruptions of a fix, returning a set of failing inputs. The loop at lines 7–11 samples inputs from I, ignoring those that trigger the original bug. We use the fact that P_b fails the assertion φ to discover such inputs. In comparing P_f and P_b on those inputs, we do not assume that P_b has only the one bug b and works correctly for all other inputs; instead, we simply assume that we can consider each bug in isolation. Further, outside of $\tilde{i}_{,b}$ P_b usefully approximates the ideal behavior of P; when P_b is a release, deployed version of a program, it presumably passed regression testing.

2.4 Empirical Evaluation

Our evaluation objective is two-fold: to demonstrate the feasibility and utility of our approach, and to differentiate WP_d from WP. First, we describe our implementation, our computing environment and how we selected our test suite. We then show that Fixation detects the bad fix in our motivating example, as well as five others. We compare WP_d to WP by showing how WP_d accumulates path predicates, and thus subsets of the true bug-triggering input domain $\tilde{i}_b$, as a function of its parameter d. Finally, we close by describing how a developer might use Fixation to discover bad fixes and instead commit good ones.

2.4.1 Implementation

Many tools and techniques exist for detecting the disruption of a fix, so we focused our implementation on determining fix coverage. We used the WALA framework [42] to extract the concrete buggy path induced by a bug-triggering input and build a CFG, from which we extract candidate paths using Algorithm 1. The extracted concrete path is the sequence of basic blocks traversed during a particular execution of the program. ICFG_n is produced by traversing the CFG of each method and unrolling each loop the number of times it iterated in the concrete path and an additional x times, according to an unrolling parameter x; thus, $n = x + y$, where y is a loop that iterated the most times in the concrete path. The unrolling parameter allows Fixation to explore paths that loop more often than the concrete path specifies. We then run our implementation of Algorithm 1 over this ICFG_n and feed each resulting path predicate to the SMT-solver CVC3 [4], keeping only those that are satisfiable.

Java PathFinder is the Swiss army knife of Java verification [87]. Fixation uses JPF's Symbc component to symbolically execute the fixed program, using the weakest precondition produced above as the precondition. If, given this precondition, the assertion fails in the fixed program, Symbc generates a concrete input that causes the failure and returns it as a counterexample.

2.4.2 Experimental Setup

We built and ran our evaluations on a Dell XPS 630i with 2.4GHz QuadCPU processors and 3.2 GB of Memory, running Ubuntu 8.04 with kernel Linux2.6.24-21-generic. Fixation is built for and runs on JRE 6.

Although we found many bad fixes in the three projects we explored, most of them were not fit for evaluation: either the bug comments were unclear or no fix was uploaded. No convention appears to govern the use of Bugzilla. Some programmers tend to write detailed logs of their fix activity and upload their fix, but most do not. We selected the first six bugs we found whose comments proved the existence of bad fix and that had an attached fix. Five of the bad fixes are from Rhino and `MultiTask` is from Ant[3].

Currently, Fixation does not directly work on the original, unmodified code, since both WP_d and Symbc (JPF v4.1) support only Java programs consisting of statements and expressions that use only boolean and integer variables. Thus, we first manually sliced away all code not related to the bugs that caused the bad fixes. We then transformed the code into an integer program that, given the same inputs, traverses the same paths as the sliced version of the original program. Since Fixation simply ignores non-integer language constructs, like function calls and field or array accesses, we left them in place to approximate the original program as closely as possible. We manually transformed the conditionals in the original code into integer conditionals. To rewrite `fun instanceof NoSuchMethodCall`, we introduced the integer variable `fun_int` and the integer constant `NoSuchMethodCall_int`, then replaced the original conditional with the conditional `fun_int == NoSuchMethodCall_int`. We added logic as necessary so that `fun_int` correctly tracked the type of original object `fun`. At each point the bug manifested itself in the original program, we added an assertion.

Our principal goal was to faithfully retain the inherent complexity of the program's logic

[3] The sliced and transformed versions of the bad fixes are available at `http://wwwcsif.cs.ucdavis.edu/~gu/bugs.htm`.

Name	Loc		Nodes		Arcs		CC	
	P	P_i	P	P_i	P	P_i	P	P_i
`NoSuchMethod`	60	65	60	64	70	75	12	13
`MultiTask`	23	36	51	62	54	68	5	8
`Substring`	8	17	10	16	10	18	2	4
`NativeErr`	9	20	10	18	10	21	2	5
`Loop`	34	42	42	46	46	51	6	7
`PathExp`	114	133	103	119	124	147	23	30

Table 2.1: Bad fix CFG complexity.

through the transformation. Table 2.1 presents evidence that we succeeded; it shows the raw lines of code, the nodes and arcs in the ICFG, and the Cyclomatic complexity (CC) of ICFGs before (P) and after P_i transformation. For a program with one exit point, Cyclomatic complexity equals the number of decision points in the program plus one [61]; we increase it in all six cases.

2.4.3 Experimental Results

Table 2.2 details the results of evaluating the six examples. The second column briefly describes each bad fix. The third column contains the counterexample Fixation reports. The fourth column d presents the edit distance used to construct the candidate paths. C denotes the number of paths Fixation explored before determining the fix to be bad. As a point of reference for C, P_a is the total number of paths. Time records the time-to-completion.

Neglected Execution Paths The first four bad fixes in Table 2.2 are all due to a programmer's ignorance of potential buggy paths. Fixation detected the buggy paths missed by each of these bad fixes while exploring a limited number of paths. Once it detected a bad fix, Fixation reported a counterexample to help programmers refine their fixes. Since the bad fixes `MultiTask`, `Substring` and `NativeErr` all exhibit essentially the same symptoms as `NoSuchMethod`, we describe only `NoSuchMethod`.

Name	Description	Counterexample	d	C	P_a	Time (s)
`NoSuchMethod`	Neglect an input.	`fun == NoSuchMethodShim && op == Icode_TAIL_CALL`	0	1	30	0.664
`MultiTask`	Forget targets' size can be zero.	`type == VECTOR && size == 0`	3	32	48	1.637
`Substring`	Miss a condition.	`i == 1 && sT == SUB_NULL`	2	3	3	0.598
`NativeErr`	Miss handling an exception.	`type == NativeError`	2	4	5	0.938
`Loop`	Fail to detect bugs in loop.	`i_c == 6 && m_l = 3`	5	354	∞	42.542
`PathExp`	Miss bugs on different paths.	`tG == -10000 && cT == T_PRI && cI == T_NULL && op == T_SHNE && sC == T_TRUE`	27	234	1021	8.308
`PathExplosion2`	Miss bugs on different paths.	`tG == -10000 && cT == T_PRI && cI == T_NULL && op == T_SHEQ && sC == T_FALSE && ...`	6	237	1021	8.518

Table 2.2: Fixation results.

Figure 2.4.1 lists the original buggy code that required three fixes sin Section 2.2. The original bug-triggering input `fun == NoSuchMethodShim && op != Icode_TAIL_CALL` reminded the programmer that there was no block for `NoSuchMethodShim`, so the programmer committed the first fix in Figure 2.2.1. Fixation took the initial bug-triggering input and buggy code, ran with distance d set to zero, and discovered the bug-triggering input domain `fun != InterpretedFun && fun != Continuation && fun != IdFunctionObject`. Fixation then symbolically executed the first fixed program, imposing that predicate together with the set predicate of the program's input domain as the initial precondition[4]. Given that precondition, Fixation reached and implied an assertion

[4]Without the predicate of the program's input domain, Symbc may generate nonsensical inputs. Assume that a program's input domain is $x > 0 \wedge x < 10$ and the computed weakest precondition is $x > 5$. With only $x > 5$, Symbc

```
1  instructionCounting++;
2  ...
3  stackTop -= 1 + indexReg;
4  if( fun == InterpretedFun ) {
5    return processInterFun();
6  }
7  if( fun == Continuation ) {
8    return processCon();
9  }
10 if( fun == IdFunctionObject ) {
11   return processIdFunObj();
12 }
13 ...
14 assert( false ); // Should never execute.
```

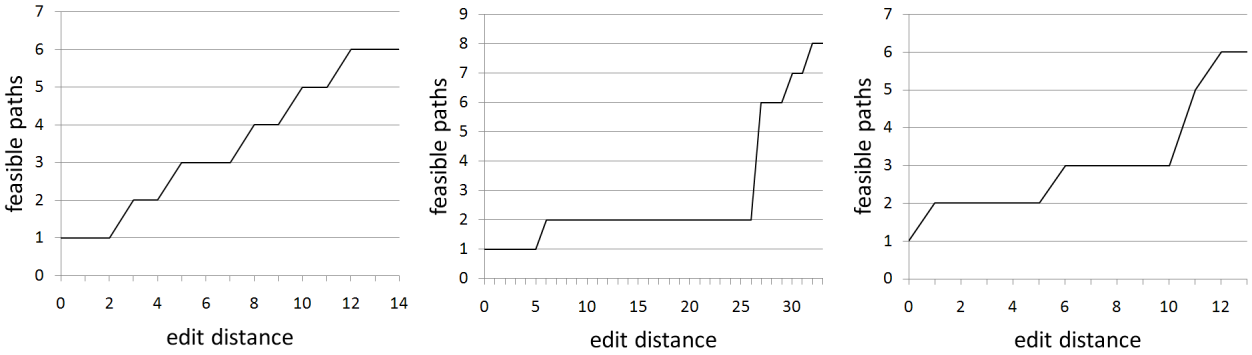
Figure 2.4.1: Sliced integer version of `NoSuchMethod`.

failure, then reported the fix bad and returned the counterexample `fun == NoSuchMethodShim && op == Icode_TAIL_CALL`.

A Bad Fix in a Loop Standard WP needs loop invariants, which are difficult to derive in general. Current tools usually adopt heuristics at the cost of sacrificing precision [11, 21]. Here, we show how WP_d tackles the loop problem. Two bugs lurk in Figure 2.4.3 — one outside the loop and the other inside. The initial bug-triggering input `i_c == 5 && m_1 == 2` exposes only the bug outside the loop. The first committed fix resolved only this bug. We ran Fixation with $d = 5$ on this first fix. Fixation explored 354 candidate paths, most of which were unsatisfiable. For example, computing WP on the path that takes the true branch inside the for loop in its 4th iteration generates the unsatisfiable clause `i==3 && i>4`. Disjuncting predicates from feasible paths, Fixation reported the bug-triggering input domain `(i_c == 5 && m_l > 2) || (i_c == 5 && m_l <= 2) || (i_c == 6 && m_l > 2) || (i_c == 6 && m_l <= 2)`. Given this predicate as its precondition, Fixation output the counterexample `i_c == 6 && m_l == 3`.

In essence, WP_d is unaware of loops. The candidate paths over which WP_d computes the

could generate the illegal input $x = 20$.



(a) `Loop` (b) `PathExp 1` (c) `PathExp 2`

Figure 2.4.2: Feasible paths as a function of edit distance.

weakest precondition are all simple paths, drawn from an ICFG$_n$. Figure 2.4.2a shows the number of feasible path predicates WP$_d$ discovered as a function of the edit distance. It illustrates that WP$_d$ can produce useful results when given limited resources. `Loop` has infinite paths and therefore infinite feasible paths, which we cut off at $d = 14$. By way of comparison, we ran this example using ESC/Java2 and JPF Symbc, without our path restriction. Using its default settings[5], ESC/Java2 failed to report a potential postcondition violation. Given the domain restriction we inferred in our first phase, our backward symbolic execution from the assertion failure, Symbc does reach the assertion failure, but at the cost exploring all loop iterations up to the failure, as opposed to only those iterations within d of the failing iteration; more importantly, Symbc continued searching until it reached loop iteration $1,000$, when we killed it.

Path Explosion WP$_d$ balances scalability and coverage via its edit distance parameter d. `PathExp` illustrates how Fixation detects potential bugs even when considering subsets of the potentially feasible paths. `PathExp` occurred because Rhino failed to ensure `(undefined === null)` evaluated to `false` as required by the ECMA (JavaScript) specification. In Figure 2.4.4, six nested bugs are distributed in different paths. From the initial bug-triggering input, the programmer discovered and fixed only bugs 1 and 2. We ran Fixation on `PathExp`, gradually increasing d as

[5]ESC/Java2 does not support assertions, so we translated each assertion into a postcondition annotation.

```
 1  private static final int M_T = 2;
 2  private static final int I_L = 4;
 3  ...
 4  public AdapterClass createAdapterClass(
 5       String adapterName, Scriptable ins[],
 6       int adapterType, int i_c, int m_l) {
 7    ...
 8    assert( i_c <= I_L );
 9    ...
10    for( int i = 0; i < i_c; i++ ) {
11       assert( i <= I_L && m_l <= M_T );
12       ...
13    }
14    ...
15  }
```

Figure 2.4.3: Sliced integer version of `Loop`.

shown in Figure 2.4.2b. As the figure depicts, WP_d finds no additional paths until $d = 27$. Exploring Figure 2.4.4, we find that bugs 3–6 are all in the `else` block of the first `if` statement, with conditional `( tG == -1 )`. A path that traverses any of bugs 3–6 shares few nodes with the original buggy path, which traverses bugs 1 and 2. At $d = 27$, Fixation finds the new feasible predicate `tG != -1 && c_T == T_PRI && c_I == T_NULL && op == T_SHNE && sC == T_TRUE` after exploring 234 paths. Armed with this predicate, Fixation can symbolically execute the first fixed program and report a counterexample exposing bugs 3 and 4.

Perhaps, as usual, the programmer was harried and rushing. In any case, he committed a fix that corrected only bugs 1 and 2. Had the programmer run Fixation, he would have been aware of the counterexample that his fix failed to handle. Figure 2.4.2c shows the result of running Fixation with increasing d on the partially fixed program. At $d = 5$, Fixation detects two new feasible weakest preconditions after exploring 237 paths. These predicates generate two counterexamples that identify the remaining two bugs. WP_d offers a flexible, principled, and resource-judicious way to search paths near a bug-triggering path.

```
1  if ( tG == -1 ) {
2    if ( c_T == T_PRI && c_I == T_NULL ) {
3      //bug 1
4      assert( op!=T_SHEQ && op!=T_SHNE );
5      ... //assignments to op
6      //bug 2
7      assert( op!=T_SHEQ && op!=T_SHNE );
8      ...
9    }
10 } else {
11   if ( c_T == T_PRI && c_I == T_NULL ) {
12     ...
13     //bug 3
14     assert( op!=T_SHEQ && op!=T_SHNE );
15     ... //assignments to op
16     //bug 4
17     assert( op!=T_SHEQ && op!=T_SHNE );
18     ...
19     if ( op == T_EQ || op == T_SHEQ ) {
20       if ( sC==T_FALSE ) {
21         markLabel(popGOTO, popStack);
22         addByteCode(ByteCode_POP);
23         //bug 5
24         assert(op!=T_SHEQ && op!=T_SHNE);
25       }
26       ...
27       //bug 6
28       assert( op!=T_SHEQ && op!=T_SHNE );
29     }
30   }
31 }
```

Figure 2.4.4: Sliced integer version of `PathExplosion`.

2.4.4 Threats to Validity

Fixation's edit distance parameter d allows its user to trade-off performance against the completeness of Fixation's approximation of the bug-triggering input domain. Determining the optimal setting of d to obtain a better result is an interesting problem. The paths WP_d traverses depends on d as well as the structure of the CFG. Gradually increasing d until detecting interesting paths or exceeding a resource threshold, such as time or number of paths explored, appears to be a good heuristic.

Fixation is currently not optimized. Caching the predicates of already explored paths would avoid redundant computation. Tabulating function summaries for reuse can also cache WP computations. Adopting lightweight predicate on-the-fly feasibility checking, la Snugglebug [11], might remove infeasible paths at an earlier stage.

We model bugs as assertion failures. Thus, Fixation's applicability depends on assertions being available, inferred, or written by a programmer. In many cases, obtaining such an assertion is trivial (as in thrown, fatal, exceptions). In others, it can be difficult and is an external threat to the validity of our approach: an empirical study to investigate and classify those cases would help users know when Fixation is and is not practical. As with any manual study, our results may exhibit selection bias. A larger empirical study would also help in showing the technique generalizes.

We inherit our limitation to integer programs from our symbolic execution components. This restriction makes the values of d we report optimistic as we operate on slightly smaller, sliced, versions of the original program. Thus, our reliance on slicing is both a construct and, because slicing limits the applicability and scalability of our approach, an external, threat to validity. As the state-of-the-art in symbolic execution improves (e.g. via techniques such as delta-execution [15]), so will the applicability of our approach. Nonetheless, the evaluation results are encouraging. Fixation detects bad fixes and reports counterexamples that can help programmers realize the

limitation of particular fix. WP_d has shown itself to be a promising approach to managing the path-explosion problem and side-stepping the loop invariant problems.

2.5 Related Work

Our work is the first to offer a systematic methodology for assessing the quality of a bug fix. It is related to the large body of work on software testing and analysis. This section summarizes the most closely related efforts. We divide related work into three categories: practical computation of weakest precondition, automatic test input generation, and studies of bug fixes and code changes.

2.5.1 Practical Computation of WP

Dijkstra's weakest precondition [17] has been extended to check the correctness of object-oriented programs. ESC/Java pioneered its use in Java [21]. ESC/Java requires user-defined annotations to specify the precondition, postcondition and invariants. By checking the validity of the verification condition generated from guarded commands, ESC/Java warns of potential bugs such as postcondition violations or null pointer dereferences. ESC/Java's checking is modular so it relies on user annotation for procedure calls. To handle loops, it heuristically iterates 1.5 times.

Snugglebug [11] presents an interprocedural WP technique. It introduces directed call graph construction, generalization, and a current search heuristic to improve the performance and precision. Polymorphism means that Java programs face the dynamic-dispatch problem when encountering function calls; directed call graph construction helps find the exact callee without exhaustive search. Generalization enhances function summary reuse using tabulation. The current search heuristic of Snugglebug prioritizes paths with less looping or call depth.

Path-based WP computation complements WP computation over an entire program. Applying counterexample-driven refinement, BLAST [37] and SLAM [3] check an abstract path to see if it corresponds to a concrete trace of the program reaching an error state. He and Gupta proposed

path-based weakest precondition to locate and correct an erroneous statement in a function [36]. Their path-based weakest precondition is similar to our WP_d when $d = 0$. We have assumed a fix at least covers the original bug-triggering input, so single-path approaches may not handle the bad fix problem. Our approach generalizes path-based WP by parameterizing the distance budget d and allowing arbitrary non-zero distances.

Our WP_d filters the paths over which standard WP computes. Instead of computing WP on all paths, WP_d approximates the true bug-triggering input domain by computing WP on paths that are close to the original buggy path. Users control the performance and coverage of WP_d through its edit distance parameter. As d increases, WP_d generates more paths and takes more time to compute. Another insight of WP_d is that computing the weakest precondition of simple paths near a known concrete path is precise and does not rely on heuristics: the concrete path specifies how to resolve a dynamic dispatch target or determine how often to traverse a loop.

2.5.2 Automatic Test Input Generation

Automatic test input generation is an active area of research. Dozens of techniques and tools have been proposed. For brevity, we highlight only some of the closely related work.

CUTE [82] and DART [24] combine concrete and symbolic execution to generate test inputs for C programs. Java PathFinder (JPF) [87] performs generalized symbolic execution to generate inputs for Java programs. We use Symbc from JPF [72] to generate counterexamples. We impose preconditions to guide the symbolic execution: Given the set predicate of an approximation of the bug-triggering input domain as the precondition, we ask whether the fixed program can reach the assertion failure or not. This precondition restricts the search space and enhances the performance of Symbc. If Symbc outputs a concrete input that causes the assertion to fail, we deem the fix bad and return that input as a counterexample. Csallner *et al.*'s work [13] combines static checking and concrete test-case generation. They perform random testing using the counterexample predicate

generated by ESC/Java. Random testing may miss some paths. We symbolically execute all paths under imposed precondition. Beyer *et al.* extend BLAST to generate testcases by finding inputs that satisfy predicates collected from all paths in the program [7]. We also collect predicates to generate inputs, but for a different purpose: Beyer *et al.* seek to exhaustively test one program, while we use symbolic execution on a buggy and a fixed version of a program to evaluate fixes.

2.5.3 Bug Fixes and Code Changes

Research on bug fixes mainly falls into two camps: mining software repositories and empirical study. BugMem [48] mines bug fix history to predict potential bugs. Kim *et al.* predict faults by consulting bug and fix caches they build [49]. Their work shows that bugs exhibit locality and inspired our design of WP_d. Anvik *et al.* applied machine learning to find programmers who should be responsible for the fix [1]. Weiss *et al.* predict fixing effort needed for a particular bug [91] and Śliwerski *et al.* proposed a way to locate code fixed using repository mining [83]. Most of the work in this domain is probabilistic and may not be precise. We propose a sound analysis for evaluating bug fixes: every bad fix we detect is an *actual* bad fix.

Code change is fundamental to software development. Ryder and Tip propose change impact analysis to find failure inducing changes and thus judge the quality of change [79, 93]. We consider a subset of the changes they consider, specifically bug fixes. Change impact analysis applies delta debugging on a sequence of atomic changes drawn from the comparison of two versions to locate suspicious changes, while we combine our distance-bounded weakest precondition with symbolic execution to judge the quality of a fix. Differential symbolic execution [68] seeks to precisely characterize the differences between two program versions. DSE exploits abstract summaries of code that is common between two program versions, as the effects of such code do not contribute to differences. In contrast, Fixation seeks to identify not only issues of disruption, but also coverage: a subset of inputs for which both versions behave the same, *viz.* by manifesting a

particular bug.

McCamant and Ernst compare operational abstractions of components and their potential replacements to predict the safety of a component upgrade [62]. Our approach is more fine-grained: assertions need not refer to the modified component as operational abstractions must. Their focus is different from ours: they seek to verify that a newer component will behave as expected under the conditions its predecessor was exposed to, and we seek to verify that a component that does indeed behave differently (due to a bug fix) does so safely (*i.e.* without disruption) and completely (*i.e.* handling all of $\tilde{i}_b$).

Regression testing validates modified software to ensure changed code has not adversely affected unchanged code [34, 67, 77, 80]. The cost of regression testing has been extensively studied and shown that test suite size can be reduced without compromising safety [26, 76]. Currently, we combine regression and random testing to check the disruption of a fix.

2.6 Discussion and Future Work

When run on buggy and allegedly fixed versions of a program, Fixation reports a new bug-triggering input drawn from an under-approximation of the true bug-triggering input domain. This new bug-triggering input is a counterexample to the implicit assertion that the fix is good. Fixation can miss bad fixes if it fails to explore a buggy path, but it is sound when it asserts that a fix is bad: every counterexample Fixation reports is certain to cause the fixed program fail the assertion.

We have introduced the bad fix problem and provided empirical evidence of its existence in real projects. We have formalized the bad fix problem and proposed an approach that combined our distance-bounded weakest precondition with symbolic execution to evaluate fixes and detect bad ones. We implemented our idea in a prototype Fixation and evaluated it: Fixation was able to detect bad fixes extracted from real-world programs.

In the future, we plan to extend Fixation to support more language features in Java and make

it applicable to real code. We intend to implement the optimizations mentioned. Unit testing is an immediate application of Fixation. A failed testcase is an ideal original bug-triggering input for Fixation. Self-contained assertions in a test suite will allow Fixation to work directly on the code and obviate manually constructing and inserting the assertion. The fact that the distribution of code tested by unit testing is relatively local is likely to be a good fit for WP_d.

Chapter 3

OSCILLOSCOPE: Reusing Debugging Knowledge via Trace-based Bug Search

Some bugs, among the millions that exist, are similar to each other. One bug-fixing tactic is to search for similar bugs that have been reported and resolved in the past. A fix for a similar bug can help a developer understand a bug, or even directly fix it. Studying bugs with similar symptoms, programmers may determine how to detect or resolve them. To speed debugging, we advocate the systematic capture and reuse of debugging knowledge, much of which is currently wasted. The core challenge here is how to search for similar bugs. To tackle this problem, we exploit semantic bug information in the form of execution traces, which precisely capture bug semantics. This research introduces novel tool and language support for semantically querying and analyzing bugs.

We describe OSCILLOSCOPE, an Eclipse plugin, that uses a bug trace to exhaustively search its database for similar bugs and return their bug reports. OSCILLOSCOPE displays the traces of the bugs it returns against the trace of the target bug, so a developer can visually examine the quality of the matches. OSCILLOSCOPE rests on our bug query language (BQL), a flexible query language over traces. To realize OSCILLOSCOPE, we developed an open infrastructure that

consists of a trace collection engine, BQL, a Hadoop-based query engine for BQL, a trace-indexed bug database, as well as a web-based frontend. OSCILLOSCOPE records and uploads bug traces to its infrastructure; it does so automatically when a JUnit test fails. We evaluated OSCILLOSCOPE on bugs collected from popular open-source projects. We show that OSCILLOSCOPE accurately and efficiently finds similar bugs, some of which could have been immediately used to fix open bugs.

3.1 Introduction

Millions of bugs have existed. Many of these bugs are similar to each other. When a programmer encounters a bug, it is likely that a similar bug has been fixed in the past. A fix for a similar bug can help him understand his bug, or even directly fix his bug. Studying bugs with similar causes, programmers may determine how to detect or resolve them. This is why programmers often search for similar, previously resolved, bugs. Indeed, even finding similar bugs that have not been resolved can speed debugging.

We theorize that, in spite of the bewildering array of applications and problems, limitations of the human mind imply that a limited number of sources of error underlie bugs [51]. In the limit, as the number of bugs in a bug database approaches all bugs, an ever larger proportion of the bugs will be similar to another bug in the database. We therefore hypothesize that, against the backdrop of all the bugs programmers have written, unique bugs are rare.

Debugging unites detective work, clear thinking, and trial and error. If captured, the knowledge acquired when debugging one bug can speed the debugging of similar bugs. However, this knowledge is wasted and cannot be reused if we cannot search it. The challenge is to efficiently discover similar bugs. To answer this challenge, this research employs traces to precisely capture the semantics of a bug. Informally, an execution trace is the sequence of operations a program performs in response to input. Traces capture an abstraction of a program's input/output behavior.

A bug can be viewed as behavior that violates a program's intended behavior. Often, these violations leave a footprint, a manifestation of anomalous behavior (Engler *et al.* [20]) in a program's stack or execution trace.

This research introduces novel tool and language support to help a programmer accurately and efficiently identify similar bugs. To this end, we developed OSCILLOSCOPE, an Eclipse plugin, for finding similar bugs and its supporting infrastructure. At the heart of this infrastructure is our bug query language (BQL), a flexible query language that can express a wide variety of queries over traces. The OSCILLOSCOPE infrastructure consists of 1) a trace collector, 2) a trace-indexed bug database, 3) BQL, 4) a query engine for BQL, and 5) web and Eclipse user interfaces. OSCILLOSCOPE is open and includes the necessary tool support to facilitate developer involvement and contribution.

The OSCILLOSCOPE database contains and supports queries over both stack and execution traces. Stack traces are less precise but cheaper to harvest than execution traces. We quantify this precision trade-off in Section 3.4.4. When available, stack traces can be very effective, especially when they capture the program point at which a bug occurred [6, 81]. Indeed, a common practice when bug-fixing is to paste the error into a search engine, like Google. Usually, the error generates an exception stack trace. Anecdotally, this tactic works surprisingly well, especially with the errors that novices make when learning an API. OSCILLOSCOPE generalizes and automates this practice, making systematic use of both execution and stack traces.

To validate our hypothesis that unique bugs are rare, we collected 877 bugs from the Mozilla `Rhino` and Apache `Commons` projects. We gave each of these bugs to OSCILLOSCOPE to search for similar bugs. We manually verified each candidate pair of similar bugs that OSCILLOSCOPE reported. If we were unable to determine, within 10 minutes, that a pair of bugs was similar, *i.e.* knowing one bug a programmer could easily fix the other, we conservatively deemed them dissimilar. Using this procedure, we found similar bugs comprise a substantial portion of these bugs, even against our initial database: $\frac{273}{877} \approx 31\%$. OSCILLOSCOPE finds duplicate bug reports

as a special case of its search for similar bugs; while duplicates comprised 74 of the 273 similar bugs, however, the majority are *nontrivially similar*. These bugs are field bugs, not caught during development or testing, and therefore less likely to be similar, a fact that strengthens our hypothesis. When querying unresolved bugs against resolved bugs in the `Rhino` project, OSCILLOSCOPE matches similar bugs effectively, using information retrieval metrics we precisely define in Section 3.4.1. Of the similar bugs OSCILLOSCOPE returns, 48 of the could have been immediately used to fix open bugs.

Finding bug reports similar to an open, unresolved bug promises tremendous practical impact: it could reuse the knowledge of the community to speed debugging. Linus' Law states "given enough eyeballs, all bugs are shallow." An effective solution to the bug similarity problem will help developers exploit this precept by allowing them to reuse the eyes and minds behind past bug fixes. OSCILLOSCOPE has been designed and developed to this end.

We make the following main contributions:

- We articulate and elaborate the vision that most bugs are similar to bugs that have already been solved and take the first steps toward a practical tool built on traces that validates and shows the promise of this vision;
- We present OSCILLOSCOPE a tool that uses traces to find similar bugs and reuse their debugging knowledge to speed debugging;
- We have developed an open infrastructure for OSCILLOSCOPE, available at `http://bql.cs.ucdavis.edu`, comprising trace collection, a trace-indexed bug database, the bug query language BQL, a Hadoop-based query engine, and web-based and Eclipse plugin user interfaces; and
- We demonstrate the utility and practicality of our approach via the collection and study of bugs from the Apache `Commons` and Mozilla `Rhino` projects.

```
1  DynaBean myBean = new LazyDynaBean();
2  myBean.set("myDynaKey", null);
3  Object o = myBean.get("myDynaKey");
4  if ( o == null )
5          System.out.println(
6                  "Expected␣result."
7          );
8  else
9          System.out.println(
10                 "What␣actually␣prints."
11         );
```

Figure 3.2.1: When a key is explicitly bound to `null`, `LazyDynaBean` does not return `null`.

3.2 Illustrating Example

Programmers must often work with unfamiliar APIs, sometimes under the pressure of a deadline. When this happens, a programmer can misuse the API and trigger cryptic errors. The following section describes a real example.

Java programmers use getter and setter methods to interact with Java beans. To handle dynamic beans whose field names may not be statically known, Java provides `Reflection` and `Introspection` APIs. Because these APIs are hard to understand and use, the Apache `BeanUtils` project provides wrappers for them. `BeanUtils` allows a programmer to instantiate a `LazyDynaBean` to set and get value lazily without statically knowing the property name of a Java bean, as on line 2 of Figure 3.2.1. When an inexperienced programmer used `LazyDynaBean` in his project, he found, to his surprise, that, even though he had explicitly set a property to `null`, when he later retrieved the value from the property, it was not `null`. Figure 3.2.1 shows sample code that exhibits this bug: executing it prints "What actually prints.", not "Expected result.".

Since this behavior was quite surprising to him, the programmer filed bug `BeanUtils-342` on March 21, 2009. Five months later, a developer replied, stating that the observed behavior is the intended behavior. In Figure 3.2.2, `LazyDynaBean`'s `get` method consults the internal map `values` on line 9. If the result is `null`, the `get` method first calls the method `createOtherProperty`, which

```
1  public Object get(String name) {
2          if (name == null) {
3                  throw new IllegalArgumentException(
4                          "No property name specified"
5                  );
6          }
7
8          // Value found
9          Object value = values.get(name);
10         if (value != null) {
11                 return value;
12         }
13
14         // Property doesn't exist
15         value = createProperty(name,
              dynaClass.getDynaProperty(name).getType());
16
17         if (value != null) {
18                 set(name, value);
19         }
20
21         return value;
22 }
```

Figure 3.2.2: `LazyDynaBean.get(String name)` from revision r295107, Wed Oct 5 20:35:31 2005.

by default calls `createProperty` to instantiate and return an empty object. In the parameter list of `createOtherProperty`, `get` calls `getDynaProperty`, which returns `Object.class` on a name bound to `null`. He did, however, suggest a workaround: subclass `LazyDynaBean` and override its `createOtherProperty` method to return `null` when passed `Object.class` as its type parameter. This in turn would cause `LazyDynaBean.get()` to return `null` at line 26, the desired behavior.

How could OSCILLOSCOPE have helped the programmer solve this problem? Assuming the OSCILLOSCOPE database had been populated with traces from the `BeanUtils` project, a programmer would use OSCILLOSCOPE to look for bugs whose traces are similar to the trace for her bug, then return their bug reports. Then she would examine those bug reports to look for clues to help her understand and fix her bug. Ideally, she would find a fix that she could adapt to her bug.

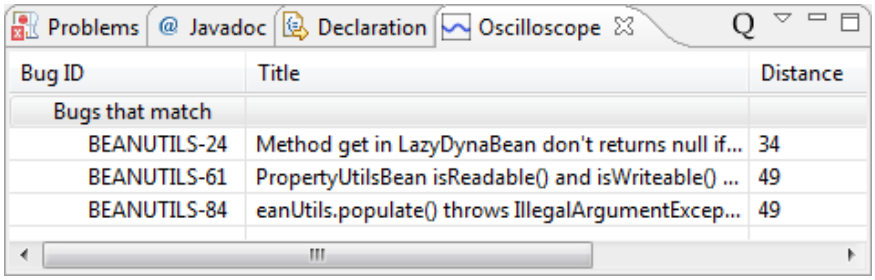


Figure 3.2.3: OSCILLOSCOPE returns bug reports similar to `BeanUtils-342`.

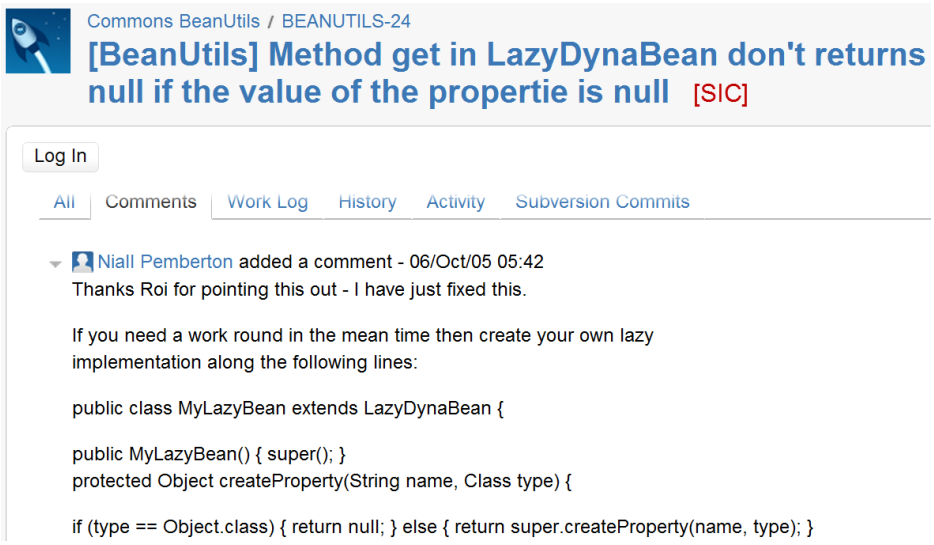


Figure 3.2.4: Snapshot of the bug report for `BeanUtils-24`.

Bug `BeanUtils-342` is the actual bug whose essential behavior Figure 3.2.1 depicts. To use OSCILLOSCOPE to search for bugs similar to `BeanUtils-342`, a developer can first issue a predefined query. When a developer does not yet know much about their current bug, a predefined query that we have found to be particulary effective is the "suffix query". This query deems two bugs to be similar when the suffixes of their traces can be rewritten to be the same; its effectiveness is due to the fact that many bugs terminate a program soon after they occur. When a developer specifies the suffix length and edit distance and issues the suffix query to search for bugs similar to `BeanUtils-342`, OSCILLOSCOPE returns the bug reports in Figure 3.2.3. The first entry is `BeanUtils-24`, where the `get` method of `LazyDynaBean` did not return `null` even when the property was explicitly set to `null`.

OSCILLOSCOPE executes the suffix query by computing the edit distance of the suffix of `BeanUtils-342`'s trace against the suffix of each trace in its database. Here is the tail of the method call traces of `BeanUtils-342` and `BeanUtils-24`, the closest bug OSCILLOSCOPE found:

BeanUtils-342	BeanUtils-24
...	...
LazyDynaBean set	BasicDynaClass setProperties
LazyDynaBean isDynaProperty	DynaProperty getName
LazyDynaClass isDynaProperty	DynaProperty getName
LazyDynaClass getDynaProperty	LazyDynaBean isDynaProperty
DynaProperty getType	LazyDynaClass isDynaProperty
LazyDynaBean createProperty	LazyDynaClass getDynaProperty
LazyDynaBean createOtherProperty	LazyDynaBean get
LazyDynaBean set	LazyDynaBean createProperty
DynaProperty getType	LazyDynaBean set
LazyDynaBean class$	DynaProperty getType

Each method call in these two traces is an event; informally, OSCILLOSCOPE looks to match events, in order, across the two traces. Here, it matches the two calls to `isDynaProperty` followed by `getDynaProperty`, then the calls to `get` and `set`. Intuitively, the distance between these two traces is the number of method calls one would have to change to make the traces identical.

Figure 3.2.4 is the snapshot of the bug report of `BeanUtils-24`. The same developer who replied to `BeanUtils-342` had also replied to `BeanUtils-24` four years earlier. From his fix to `BeanUtils-24`, the fix for `BeanUtils-342` is immediate. With the help of OSCILLOSCOPE, the programmer could have solved the bug in minutes, instead of possibly waiting five months for the answer. This example shows how OSCILLOSCOPE can help a programmer find and reuse the

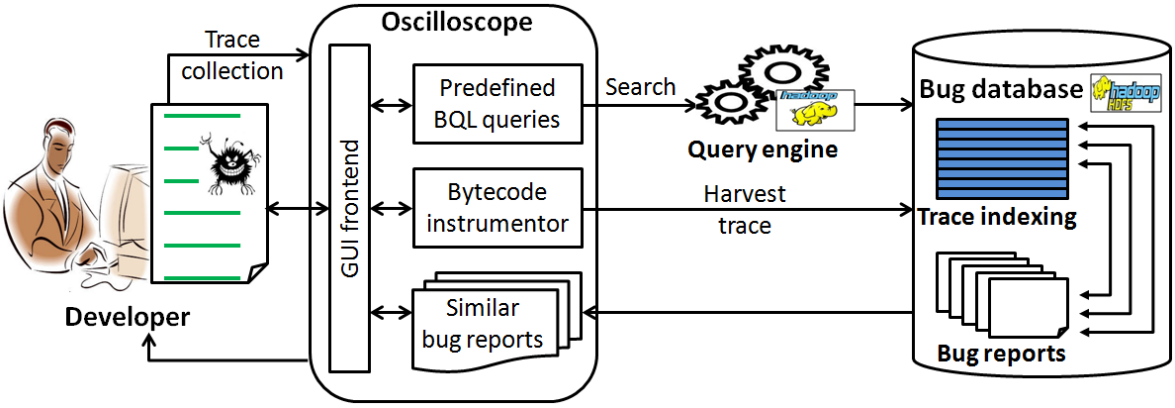


Figure 3.3.1: Debugging with the OSCILLOSCOPE Framework.

knowledge embodied in a bug report to fix an open bug.

3.3 Design and Realization of OSCILLOSCOPE

This section introduces the key components of OSCILLOSCOPE: its user-level support for trace-based search for similar bugs, its bug query language, and core technical issues we overcame to implement it.

3.3.1 User-Level Support

To support trace-based search for similar bugs, OSCILLOSCOPE must harvest traces, allow users to define bug similarity either by selecting predefined queries or by writing custom queries, process those queries to search a trace-indexed database of bug reports, display the results, and present a user interface that makes this functionality easy to use. Figure 3.3.1 depicts the architecture of OSCILLOSCOPE that supports these tasks.

Eclipse Plugin Most developers rely on an integrated development environment (IDE); to integrate OSCILLOSCOPE smoothly into the typical developer's tool chain and workflow, especially to complement the traditional debugging process, we built OSCILLOSCOPE as an Eclipse plugin.

OSCILLOSCOPE also supports a web-based user interface, described in a tool demo [29].

OSCILLOSCOPE automates the instrumentation of buggy programs and the uploading of the resulting traces. When a developer who is using OSCILLOSCOPE encounters a bug and wants to find the bug reports similar to her current bug, she tells OSCILLOSCOPE to instrument the buggy code, then re-triggers the bug. OSCILLOSCOPE automatically uploads the resulting trace to its trace-indexed database of bug reports.

Predefined Queries OSCILLOSCOPE is equipped with predefined queries; in practice, users need only select a query and specify that query's parameters, such as a regular expression over method names or an edit distance bound, *i.e.* a measure of the cost of writing one trace into another which Section 3.3.2 describes in detail. Since bugs often cause a program to exit quickly, we have found that the suffix query, (introduced in Section 3.2) which compares short suffixes of traces with a modest edit distance bound, to be quite effective. Section 3.4.1 describes how we discovered and validated this suffix query. The bulk of OSCILLOSCOPE's predefined queries, like the suffix query, have simple, natural semantics. Once the buggy trace has been uploaded, the developer can allow OSCILLOSCOPE to automatically use the last selected query to search for similar bugs and return their bug reports, or select a query herself. OSCILLOSCOPE's query engine, which is based on Hadoop, performs the search.

When a JUnit test fails, these steps occur automatically in the background: OSCILLOSCOPE instruments and reruns the test, uploads the resulting test to the database, then, by default, issues a predefined suffix query to search for similar bugs and returns the results. This feature is especially valuable as it targets bugs that occur during development and are more likely to be similar to other bugs than field bugs which have evaded regression testing, inspection and analysis to escape into deployment. To speed response time, OSCILLOSCOPE returns partial results as soon as they are available. Users refresh to see newer results. OSCILLOSCOPE can also visually compare traces to help developers explore and understand the differences and similarities between two traces.

To find the bug report with the fix to the bug in our illustrating example in Section 3.2, the

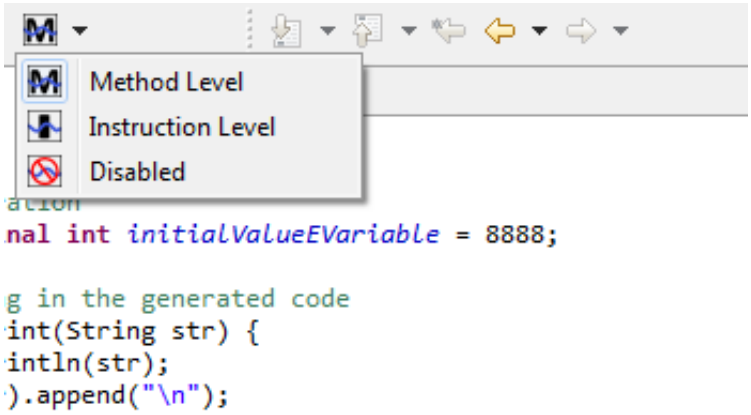


Figure 3.3.2: Selecting execution trace granularity in the OSCILLOSCOPE Eclipse plugin.

developer would direct OSCILLOSCOPE to instrument the buggy code, re-triggered the bug, then issued the default suffix query. Behind the scenes, OSCILLOSCOPE would harvest and upload the trace, then execute the query and display the resolved bug report with the relevant fix.

Trace Granularity Using the OSCILLOSCOPE, programmers can trace code at statement or method granularity, as show in Figure 3.3.2. Statement-granular traces allow OSCILLOSCOPE to support the search for *local* bugs, bugs whose behavior is contained within a single method, and facilitates matching bugs across different projects that run on the same platform and therefore use the same instruction set architecture. OSCILLOSCOPE supports coarser granularity traces, such as replacing a contiguous sequence of method call on a single class with that class name, by post-processing.

Chop Points Instrumentation is expensive, so we provide OSCILLOSCOPE provides a chop operator that allows programmers to set a *chop point*, a program point at which tracing starts or ends. A programmer can also use chop points to change the granularity of tracing from method calls to statements. By restricting tracing to occur within a pair of chop points, chopping enables the scalability of fine-grained, *i.e.* statement-granular, tracing. When debugging with OSCILLOSCOPE, a developer will resort to setting chop points to collect a partial trace for their current bug when collecting a complete trace is infeasible. Knowing where to place chop points involves

guesswork; its payoff is the ability to query the OSCILLOSCOPE database. Thus, the OSCILLOSCOPE database contains partial traces and traces that may contain events at different levels of granularity.

In a world in which OSCILLOSCOPE has taken hold, developers will routinely upload traces when they encounter a bug. To this end, we have setup an open bug database, available at our website, and welcome developers to contribute both the buggy traces and the knowledge they acquired fixing the bugs that caused them. Our tool is not just for students and open source developers. We have made the entire OSCILLOSCOPE framework privately deployable, so that companies can use OSCILLOSCOPE internally without having to share their bug database and worry about leaking confidential information.

We have posited that, when two bugs may share a root cause, this fact manifests itself as a similarity in their traces. Prior to this region of similarity, of course, the two traces might be arbitrarily different. Internally, OSCILLOSCOPE relies on BQL, its bug query language, and the insight and ingenuity of query writers to write BQL queries that isolate the essence of a bug; these queries define bug similarity and drive OSCILLOSCOPE's search for similar bugs. Thus, OSCILLOSCOPE rests on BQL, which we describe next.

3.3.2 BQL: A Bug Query Language

A bug is behavior that violates a user's requirements. The core difficulty of defining a bug more precisely is that different users, even running the same application, may have different requirements that change over time. In short, one user's bug can be another's feature. To tackle this problem, we have embraced expressivity as the central design principle of BQL; it allows writing of queries that embody different definitions of buggy behavior. To support such queries, our database must contain program behaviors. Three ways to capture behavior are execution traces, stack traces, and vector summarizations of execution traces. An execution trace is a sequence of

events that a program emits during execution and precisely captures the behavior of a program on an input. Collecting execution traces is expensive and they tend to be long, mostly containing events irrelevant to a bug. Stack traces encode execution events keeping only the current path from program entry to where a bug occurs. Finally, one can summarize an execution trace into a vector whose components are a fixed permutation of a program's method calls. For example, the summarization of trace *ABA* into a vector whose components are ordered alphabetically is $\langle 2,1 \rangle$, which discards the order of events. In Section 3.4.4 we quantify the loss of precision and recall these approaches entail. Therefore, we define bug similarity and support queries for bug data, such as a fix, in terms of execution traces.

Terminology

T denotes the set of all traces that a set of programs can generate. Each trace in **T** captures an execution of a program (We discuss our data model in more detail in Section 3.3.2). B is the set of all bugs. The bug $b \in B$ includes the afflicted program, those inputs that trigger a bug and the traces they induce, the reporter, dates, severity, developer commentary, and, ideally, the bug's fix. The function $t : B \to 2^{\mathbf{T}}$ returns the set of traces that trigger b. The set of all unresolved bugs U and the set of all resolved bugs R partition B. We formalize the ideal bug similarity in the oracle ϕ. For $b_0, b_1 \in B$,

$$\text{similar}(b_0, b_1) = \begin{cases} \text{T} & \text{if } b_0 \text{ is similar to } b_1 \text{ wrt } \phi \\ \text{F} & \text{otherwise} \end{cases} \tag{3.3.1}$$

which we use to define, for $b \in B$,

$$[\![b]\!] = \{x \in B \mid \text{similar}(b, x)\}, \tag{3.3.2}$$

the set of all bugs that are, in fact, similar to b.

For $b_0, b_1 \in B$, and the query processing engine Q,

$$\text{match}(b_0, b_1) = \begin{cases} \text{T} & \text{if } b_0 \text{ is similar to } b_1 \text{ wrt } Q \\ \text{F} & \text{otherwise} \end{cases} \tag{3.3.3}$$

which we use to define

$$[b] = \{x \in B \mid \text{match}(b, x)\}, \tag{3.3.4}$$

the set of all bugs that the query for b returns.

Ideally, we would like $[\![b]\!] = [b]$, but for debugging it suffices that we are 1) *sound*: $\text{match}(b_i, b_j) \Rightarrow \text{similar}(b_i, b_j)$, and 2) *relatively complete*: $\text{similar}(b_i, b_j) \Rightarrow \exists b_k \in [\![b_j]\!] - \{b_i\}\ \text{match}(b_i, b_k)$, for any $b_i \neq b_j$. In Section 3.4, we demonstrate the extent to which we use traces to achieve these goals.

Both $[\![b]\!]$ and $[b]$ are reflexive: $\forall b \in B, b \in [\![b]\!] \wedge b \in [b]$, which means that $\{b\} \subseteq [\![b]\!] \cap [b] \neq \emptyset$. We are often interested in bugs similar to b other than b itself, so we also define

$$U_b = [b] \cap U - \{b\} \qquad \text{unresolved bugs that match } b \tag{3.3.5}$$

$$R_b = [b] \cap R - \{b\} \qquad \text{resolved bugs that match } b. \tag{3.3.6}$$

The syntax of BQL

Figure 3.3.3 defines the syntax of BQL, which is modeled after the standard query language SQL. The query "`SELECT b FROM ALL WHERE INTERSECT?(b, "getKeySet")`" returns all bugs whose traces have a nonempty intersection with the set of all traces that call the `getKeySet` method. The clause `FROM Project1, Project2` restricts a query to bugs in `Project1` or `Project2`. The terminal `ALL` removes this restriction, and is the default when the `FROM` clause is omitted.

Predicates In addition to the standard Boolean operators, BQL provides `SUBSET?`, `INTERSECT?`,

$$
\begin{aligned}
\langle query \rangle ::= \ & \textbf{SELECT } \langle bug \rangle^{+} [\ \textbf{FROM } \langle db \rangle^{+}] \\
& \textbf{WHERE } \langle cond \rangle \ \ [\textbf{DISTANCE } \langle distance \rangle] \\
\langle db \rangle ::= \ & \texttt{X} \mid \textbf{ALL} \\
\langle cond \rangle ::= \ & \langle cond \rangle \ \textbf{\&\&} \ \langle cond \rangle \mid \langle cond \rangle \text{ —— } \langle cond \rangle \mid (\ \langle cond \rangle\ ) \\
& \mid \textbf{INTERSECT?}(\langle bug \rangle, \langle pat \rangle [, d][, n]) \\
& \mid \textbf{JACCARD?}(\langle bug \rangle, \langle pat \rangle, t[, d]) \\
& \mid \textbf{SUBSET?}(\langle bug \rangle, \langle pat \rangle [, d]) \\
\langle bug \rangle ::= \ & \texttt{Traces} \mid [\ len\ ] \langle bug \rangle \mid \langle bug \rangle [\ len\ ] \\
& \mid \textbf{PROJ}(\langle bug \rangle, \texttt{S}) \\
\langle pat \rangle ::= \ & \sigma \mid \langle bug \rangle \mid \langle pat \rangle \text{ — } \langle pat \rangle \mid \langle pat \rangle^{*} \mid (\ \langle pat \rangle\ )
\end{aligned}
$$

Figure 3.3.3: The syntax of BQL: **X** is a project; for the bug b, $\texttt{Traces} = t(b)$; σ is an event; and `S` is a set of events.

and `JACCARD?` predicates to allow a programmer to match a bug with those bugs whose traces match the pattern, when the target bug's traces are a subset of, have a nonempty intersection with, or overlap with those traces. For example, "`SELECT b FROM ALL WHERE SUBSET?(b,`$\texttt{b}_{527}$`)`" returns those bugs whose traces are a subset of b_{527}'s traces.

Traces may differ in numerous ways irrelevant to the semantics of a bug. For example, two traces may have taken different paths to a buggy program point or events with the same semantics may have different names. Concrete execution traces can therefore obscure semantic similarity, both cross-project and even within project. As a first step toward combating the false negatives this can cause, BQL allows two traces to differ in Levenshtein edit distance within a bound. The Levenshtein distance of two strings is the minimum number of substitutions, deletions and insertions of a single character needed to rewrite one string into another. The Levenshtein distance of 011 and 00 is 2 ($011 \rightarrow 01 \rightarrow 00$).

The application of edit distance to a bug's traces generates a larger set of strings. Thus, BQL adds the distance parameter d to its set predicates to bound the edit distance used to produce traces during a similarity search. Edit distance relaxes matching and can introduce false positives. To

combat this source of imprecision, the `INTERSECT?` operator also takes n, an optional constraint that specifies the minimum number of a bug's traces that must be rewritten into one of the target bug's traces. For example, assume b_{527} contains multiple traces that trigger an assertion failure and a programmer wants to search for other bugs with multiple traces. The program could use the predicate `INTERSECT?(b, b527, 50, 3)`, which forms the set of pairs of traces $t(\texttt{b}) \times t(\texttt{b}_{527})$ and is true if the members of at least 3 of these pairs can be rewritten into one another using 50 edits.

Operators When we know enough about the problem domain or salient features of our bug, we may wish to restrict where traces match. The terminal behavior of a buggy program, embodied in the suffix of its execution trace, often captures a bug's essential features. Or we may wish to consider only those traces in which the application initialized in a certain fashion and restrict attention to prefixes. Thus, BQL provides prefix and suffix operators. These operators use array bracket notation and return the specified length prefix or suffix.

A programmer may wish to project only a subset of events in a trace. For example, when searching for bugs similar to $\texttt{b}_{527}$, a developer may want to drop methods in the `log` package to reduce noise. To accomplish this task, he writes

```
SELECT bug FROM ALL WHERE SUBSET?(
    PROJ(bug, "read,write,close"), b527, 10 )
```

where `"read,write,close"` names the only methods in the trace in which we are interested.

Patterns The last line of Figure 3.3.3 defines the BQL's pattern matching syntax. Here, the terminals are either (through the $\langle bug \rangle$ the rule), $t(bug)$, the set of traces that trigger a bug, or $\sigma \in \Sigma_x$, a symbol (*i.e.* event) in a trace. Patterns that mix symbols from different event alphabets can succinctly express subsets of traces. For instance, a query writer may wish to find bugs that traverse the class c_a on the way to the method m_1 in c_b and then execute some method in the class c_c. The pattern $c_a m_1 c_c$ achieves this. As a concrete example, consider a developer who wishes

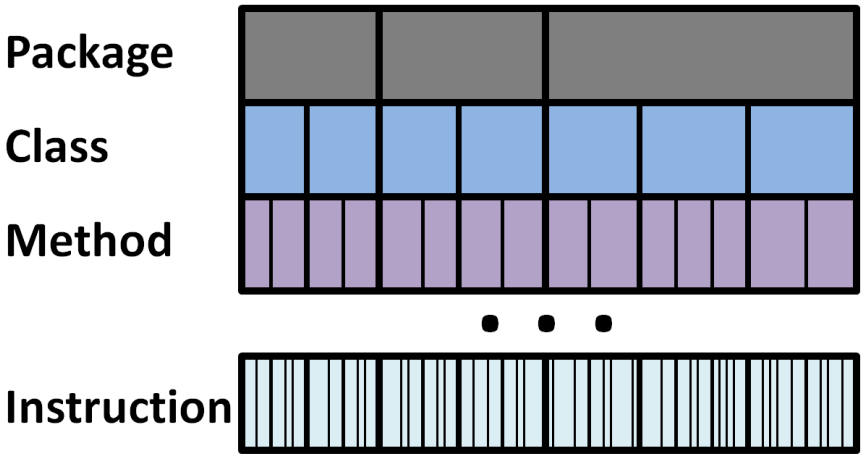


Figure 3.3.4: Hierarchy of trace event alphabets.

to find all traces that invoke methods in the class `PropertyConfiguration` (abbreviated `PC` in the query below) before invoking the method `escapeJava` in `StringEscapeUtils`; this generates the query

```
SELECT bug FROM Configuration WHERE
    INTERSECT?(bug, "PC␣escapeJava").
```

Semantics

BQL rests on a hierarchy of disjoint alphabets of execution events, shown in Figure 3.3.4. At the lowest level, execution events are instructions. An instruction symbol encodes an opcode and possibly its operands; a method symbol encodes a method's name and possibly its signature and parameters. Sequences of instructions define statements in the source language; sequences of statements define basic blocks whose sequences define a path through a method. Thus, a project defines a sequence of languages of traces defined over each alphabet in its hierarchy.

Formally, a project defines a disjoint sequence of alphabets $\Sigma_i, i \in \mathbb{N}$ where Σ_1 is the instruction alphabet. Let $\mathscr{L}(P@\Sigma)$ denote the language of traces the project P generates over the event alphabet Σ. Then, for the project P and its alphabets, each symbol in a higher level language

$$[\![\textbf{SELECT } (b_1,\cdots,b_n) \textbf{ FROM } (\texttt{db}_1,\cdots,\texttt{db}_m) \textbf{ WHERE } \texttt{cond}]\!]$$
$$= \{(b_1,\cdots,b_n) \mid (b_1,\cdots,b_n) \in [\![\texttt{cond}]\!] l \cap \bigcup_{i\in[1,m]} [\![\texttt{db}_i]\!]\}$$
$$[\![\textbf{SELECT } (b_1,\cdots,b_n) \textbf{ FROM } (\texttt{db}_1,\cdots,\texttt{db}_m) \textbf{ WHERE } \texttt{cond}$$
$$\textbf{DISTANCE } \texttt{dist}]\!]$$
$$= \{(b_1,\cdots,b_n) \mid (b_1,\cdots,b_n) \in [\![\texttt{cond}]\!][\![\texttt{dist}]\!] \cap \bigcup_{i\in[1,m]} [\![\texttt{db}_i]\!]\}$$
$$[\![\texttt{db}]\!] = \begin{cases} X & \text{if } \texttt{db} = \texttt{X} \subset \mathbf{T} \\ \mathbf{T} & \text{if } \texttt{db} = \texttt{ALL} \end{cases}$$
$$[\![\texttt{dist}]\!] \in \{h,l,u\}$$
$$[\![\texttt{cond}_1 \textbf{ \&\& } \texttt{cond}_2]\!] = \lambda\delta.[\![\texttt{cond}_1]\!]\delta \wedge [\![\texttt{cond}_2]\!]\delta$$
$$[\![\texttt{cond}_1 \texttt{ —— } \texttt{cond}_2]\!] = \lambda\delta.[\![\texttt{cond}_1]\!]\delta \vee [\![\texttt{cond}_2]\!]\delta$$
$$[\![\texttt{(cond)}]\!] = [\![\texttt{cond}]\!] = \lambda\delta.[\![\texttt{cond}]\!]\delta$$

Figure 3.3.5: Semantics of queries and Boolean operators.

$$[\![p]\!] = \mathscr{L}(p) \subseteq T \qquad [\![p^*]\!] = [\![p]\!]^*$$
$$[\![p \text{ — } p]\!] = [\![p]\!] \cup [\![p]\!] \qquad [\![(\ p\ )]\!] = [\![p]\!]$$

Figure 3.3.6: Semantics of the pattern operators.

defines a language in P's lower-level languages: $\forall\sigma \in \Sigma_{i+1}, \mathscr{L}(\sigma) \subseteq \mathscr{L}(P@\Sigma_i)$. Traces can mix symbols from alphabets of different abstraction levels, so long as there exists an instruction-level translation of the trace t such that $t \in \mathscr{L}(P@\Sigma_1)$.

The semantics of BQL is mostly standard. Figure 3.3.5 straightforwardly defines queries and the standard Boolean operators. It defines $[\![\texttt{dist}]\!]$ as a distance function and uses lambda notation to propagate its binding to the BQL's set predicates. The semantics of patterns in Figure 3.3.6 are standard; the $\mathscr{L}$, used to define pattern semantics, is the classic language operator.

Set Predicates and Edit Distance In Figure 3.3.5, $[\![b]\!] = [\![\texttt{Traces}]\!] \in 2^{\mathbf{T}}$ and the edit distance function $[\![\texttt{dist}]\!]$ has signature $\Sigma^* \times \Sigma^* \to \mathbb{N}$. The set of allowed edit distance functions is $\{h,l,u\}$. In this set, h denotes Hamming, l denotes Levenshtein (the default) and u denotes a user-specified

$$
\begin{aligned}
[\![\textbf{INTERSECT?}(b,p)]\!] &= [\![b]\!] \cap [\![p]\!] \neq \emptyset \\
[\![\textbf{INTERSECT?}(b,p,d)]\!] &= [\![b]\!] \cap_d [\![p]\!] \neq \emptyset \\
[\![\textbf{INTERSECT?}(b,p,n)]\!] &= |[\![b]\!] \cap [\![p]\!]| \geq n \\
[\![\textbf{INTERSECT?}(b,p,d,n)]\!] &= |[\![b]\!] \cap_d [\![p]\!]| \geq n \\
[\![\textbf{JACCARD?}(b,p,t)]\!] &= \frac{|[\![b]\!] \cap [\![p]\!]|}{|[\![b]\!] \cup [\![p]\!]|} \geq t \\
[\![\textbf{JACCARD?}(b,p,t,d)]\!] &= \frac{|[\![b]\!] \cap_d [\![p]\!]|}{|[\![b]\!] \cup [\![p]\!]|} \geq t \\
[\![\textbf{SUBSET?}(b,p)]\!] &= [\![b]\!] \subseteq [\![p]\!] \\
[\![\textbf{SUBSET?}(b,p,d)]\!] &= \forall x \in [\![b]\!], \exists y \in [\![p]\!] \;\; \delta(x,y) \leq d
\end{aligned}
$$

Figure 3.3.7: Semantics of the intersect, Jaccard and subset predicates.

$$
\begin{aligned}
[\![\texttt{Traces}]\!] &\in 2^{\mathbf{T}} \\
[\![\textbf{PROJ}(b,\texttt{S})]\!] &= [\![b]\!]\big|_{\texttt{S}} \\
[\![\texttt{[}\ len\ \texttt{]}b]\!] &= \{\alpha \mid \exists \beta \;\; \alpha\beta \in [\![b]\!] \wedge |\alpha| = len\} \\
[\![b\texttt{[}\ len\ \texttt{]}]\!] &= \{\beta \mid \exists \alpha \;\; \alpha\beta \in [\![b]\!] \wedge |\beta| = len\}
\end{aligned}
$$

Figure 3.3.8: Semantics of the trace operators.

edit distance function. BQL allows query writers to specify a distance function to give them control over the abstraction and cost of a query. For instance, a query writer may try a query using Hamming distance. If the results are meager, he can retry the same query with Levenshtein distance, which matches more divergent traces.

For $\delta \in \{h,l,u\}$, the function $\cap_d : 2^{\Sigma^*} \times 2^{\Sigma^*} \times \mathbb{N} \to 2^{\Sigma^*}$ is

$$
\begin{aligned}
X \cap_d Y = \{z| \;\; & z \in X, \exists y \in Y \;\; \delta(y,z) \leq d \\
& \vee z \in Y, \exists x \in X \;\; \delta(x,z) \leq d\}
\end{aligned}
\tag{3.3.7}
$$

and constructs the set of all strings in X or Y that are within the specified edit distance of an element of the other set. We use $\cap_d$ to define `JACCARD?` and `INTERSECT?` in Figure 3.3.7. For `JACCARD?`, the Jaccard similarity must meet or exceed $t \in [0,1]$; for `INTERSECT?`, the cardinality of

the set formed by $\cap_d$ must meet or exceed $n \in \mathbb{N}$.

A user-defined distance function u may be written in any language so long as it matches the required signature. One could define distance metric that reduces a pair of method traces to sets of methods then measure the distance of those sets in terms of the bags of words extracted from the method names, identifiers or comments. Alternatively, one could define a Jaccard measure over the sets of methods or classes induced by two traces, scaling the result into $\mathbb{N}$.

Trace Operators To specify the length of prefixes and suffixes in Figure 3.3.8, we use $len \in \mathbb{N}$. In the definition of **PROJ**, proj_i is the projection map from set theory and $S \subseteq \Sigma_x$, *i.e.* S is a subset of symbols from one of the event abstraction alphabets. BQL's concrete syntax supports regular expressions as syntactic sugar for specifying `S`.

3.3.3 Implementation

Four modules comprise OSCILLOSCOPE: a bytecode instrumentation module, a trace-indexed database, a query processing engine, and two user interfaces. The instrumentation module inserts recording statements into bytecode. The instrumentation module is built on the ASM Java bytecode manipulation and analysis framework. For ease of smooth interaction with existing workflows, our database has two forms: a standalone database built on Hadoop's file system and an trace-based index to URLs that point into an existing Bugzilla database. The OSCILLOSCOPE plug-in for Eclipse supports graphically comparing traces. A challenge we faced, and partially overcame, is that of allowing the comparison of traces visually regardless of their length. An interesting challenge that remains is to allow a user to write, and refine, queries visually by clicking on and selecting portions of a displayed trace. The web-based UI is AJAX-based and used Google's GWT.

Internally, traces are strings with the syntax

$$
\begin{aligned}
\langle \textit{Trace} \rangle &::= \langle \textit{Event} \rangle \mid \langle \textit{Event} \rangle \langle \textit{Trace} \rangle \\
\langle \textit{Event} \rangle &::= \langle \textit{Method} \rangle \mid \langle \textit{Instruction} \rangle \\
\langle \textit{Method} \rangle &::= \textbf{M FQClassName MethodName Signature} \\
&\mid \textbf{S FQClassName MethodName Signature} \\
\langle \textit{Instruction} \rangle &::= \textbf{I OPCODE} \langle \textit{Operands} \rangle \\
\langle \textit{Operands} \rangle &::= \textbf{OPERAND} \mid \textbf{OPERAND} \langle \textit{Operands} \rangle
\end{aligned}
$$

An example method event follows

```
M org/apache/commons/beanutils/LazyDynaClass \
getDynaProperty (Ljava/lang/String;) \
Lorg/apache/commons/beanutils/DynaProperty;.
```

Here, **M** denotes an instance method (while **S** in the syntax denotes a static method). The fully qualified class name follows it, then the method name, and finally the method signature. To capture method events, we inject logging into each method's entry. For statements, we inject logging into basic blocks. To produce coarser-grained traces, we post-process method-level traces to replace contiguous blocks of methods in a single class or package with the name of the class or package.

Query Engine The overhead of query processing lies in two places: retrieving traces and comparing them against the target trace. For trace comparison, we implemented an optimized Levenshtein distance algorithm [30]. To scale to large databases (containing millions of traces), OSCILLOSCOPE's query engine is built on top of Apache Hadoop, a framework that allows for the distributed processing of large data sets across clusters of computers. The essence of Hadoop is MapReduce, inspired by the map and reduce functions commonly used in functional program-

ming. It enables the processing of highly distributable problems across huge datasets (petabytes of data) using a large number of computers. The "map" step divides an application's input into smaller sub-problems and distributes them across clusters and "reduce" step collects the answers to all the sub-problems and combines them to form the output.

OSCILLOSCOPE's query processing is an ideal case for MapReduce, since it compares all traces against the target trace. This comparison is embarrassingly parallelizable: it can be divided into sub-problems of comparing each trace in isolation against the target trace. Each mapper processes a single comparison and the "reduce" step collects those bug identifiers bound to traces within the edit distances bound to form the final result. Section 3.4 discusses the stress testing we performed for OSCILLOSCOPE against millions of traces.

3.3.4 Extending OSCILLOSCOPE with New Queries

OSCILLOSCOPE depends on experts to customize its predefined queries for a particular project. These experts will use BQL and its operators to write queries that extract trace subsequences that capture the essence of a bug. Learning BQL itself should not be much of a hindrance to these experts, due to its syntax similarity to SQL and its reliance on familiar regular expressions. To further ease the task of writing queries, OSCILLOSCOPE visualizes the difference of two traces returned by a search and supports iterative query refinement by allowing the query writer to edit the history of queries he issued. To write effective queries, an expert will, of course, need to know her problem domain and relevant bug features; she will have to form hypotheses and, at times, resort to trial and error. The payoff for a query writer, and especially for an organization using OSCILLOSCOPE, is that, once written, queries can be used over and over again to find and fix recurring bugs. Across different versions of a project, even though methods may change names as a project evolves, OSCILLOSCOPE can still find similar bugs if their signature in a trace is sufficiently localized and enough signposts, such as method names, remain unchanged so that edit

distance can overcome the distance created by those that have changed.

Single Regex Queries `Configuration-323` occurred when `DefaultConfigurationBuilder` misinterpreted property values as lists while parsing configuration files. The reporter speculated that the invocation of `ConfigurationUtils.copy()` during internal processing was the cause. To search for bugs in the `Configuration` project that invoke the `copy()` method in the `ConfigurationUtils` class, the reporter could have issued

```
SELECT bug FROM Configuration WHERE
    SUBSET?(bug, "ConfigurationUtils.copy()").
```

The result set contains `272` and `283`, in addition to `323`. Developers acknowledged the problem and provided a workaround. Thus, the bug `323` can be solved identifying and studying `272` and `283`. These bugs predate `323`. Here, OSCILLOSCOPE found usefully similar bugs using a query based on a simple regular expression that matched a single method call.

`Lang-421` is another example of a bug for which a simple query parameterized on a regular expression over method names would have sped its resolution. In this bug, the method `escapeJava()` in the `StringEscapeUtils` class incorrectly escaped the '/' character, a valid character in Java. In Apache `Commons`, the `Configuration` project depends on `Lang`. To find out similar bugs in the `Configuration` project, we set $\alpha =$ `StringEscapeUtils.escapeJava` and issue the query

```
SELECT b FROM ALL WHERE INTERSECT?(
    PROJ(b, org/apache/commons/lang/*), 'α')
```

to search for all bugs in the database that match `pat`. This query returns four bugs. It returns `Lang-421`, the bug that motivated our search. The second bug is `Configuration-408`, where forward slashes were escaped when a URL was saved as a property name. Studying the description, we confirmed that `StringEscapeUtils.escapeJava()` caused the problem. The third bug, `Configuration-272`, concerns incorrectly escaping the ',' character; the class `StringEscapeUtils` still exhibits this problem. Manually examining the last bug, `Lang-473`, confirms that it duplicates

`Lang-421`. It is our experience with bugs like these that led us to add the simple "regex query" to our suite of predefined queries.

Limitations

Overcoming instrumentation overhead is an ongoing challenge for OSCILLOSCOPE. For example, stress-testing FindBugs revealed five-fold slowdown. Our first, and most important, countermeasure is our chop operator, described above in Section 3.3.1. In our experience, statement-granular tracing would be infeasible without it. Longer term, we plan to employ Larus' technique to judiciously place chop points [53]. Another direction for reducing overhead is to use sampling, then trace reconstruction, as in Cooperative Debugging [56]; the effectiveness of this approach depends, of course, on OSCILLOSCOPE garnering enough participation to reliably acquire enough samples to actually reconstruct traces. Currently, OSCILLOSCOPE handles only sequential programs. To handle concurrent programs, we will need to add thread identifiers to traces and explore the use of vector distance on interleaved traces.

3.4 Evaluation

This evaluation shows that OSCILLOSCOPE does find similar bugs and, in so doing, finds a generally useful class of suffix-based queries. It measures how OSCILLOSCOPE scales and demonstrates the accuracy of basing search on execution traces.

To evaluate OSCILLOSCOPE, we collected method-level traces and studied bugs reported against the Apache `Commons` (2005–2010): comprising 624153 LOC and 379163 lines of comment, and `Rhino` (2001–2010): comprising 205775 LOC and 34741 lines of comment projects. We chose these projects because of their popularity. In most cases, reporters failed to provide a test case to reproduce the bug. Even with a test case, recompiling an old version and reproducing the bug was a manual task that consumed 5 minutes on average. This explains why OSCILLOSCOPE's

trace database contains 656 of the 2390 bugs reported against Apache `Commons` and 221 of the 942 bugs reported against `Rhino`. For each bug, we recorded related information from the bug tracking system such as source code, the fix (if available), and developer comments. Our database currently contains 877 traces (one trace per bug) and its size is 43.1 MB. The minimum, maximum, mean, and variance of the trace lengths in the database are 2, 50012, 5431.1, and 6.687.

Experimental Procedure In general, we do not know $[\![b]\!]$, those bugs that are *actually* similar to each other (Equation 3.3.2). We manually approximated $[\![b]\!]$ from $[b]$, an OSCILLOSCOPE result set, in two ways. First, we checked whether two bugs shared a common *error-triggering* point, the program point at which a bug first causes the program to violate its specification. We studied *every* candidate pair of bugs that OSCILLOSCOPE reported to be similar for at most 10 minutes. For example, we deemed `Rhino-217951` and `217965` to be similar because a `Number.toFixed()` failure triggered each bug. Second, we recorded as similar any bugs identified as such by a project's developers. For example, a Rhino developer commented in `Rhino-443590`'s report that it looks like `Rhino-359651`. Given our limited time and knowledge of the projects, we often could not determine whether two bugs are, in fact, similar. When we could not determine similarity, we conservatively deemed the bugs dissimilar. This procedure discovers false positives, not false negatives. To account for false negative, we introduce the relative recall measure in Section 3.4.1 next. We discuss our methodology's construct validity in Section 3.4.5.

Using this experimental procedure, we found that similar bugs comprise a substantial portion of bugs we have collected to date: $\frac{273}{877} \approx 31\%$. 74 of the 273 similar bugs are identical, caused by duplicate bug reports; the majority, however, are nontrivially similar. Since we conjecture that the number of ways that humans introduce errors is finite and our database contains field bugs, we expect the proportion of similar bugs to increase as our database grows.

3.4.1 Can OSCILLOSCOPE Find Similar Bugs?

To show that OSCILLOSCOPE accurately finds similar bugs and to validate the utility of a default query distributed with OSCILLOSCOPE, we investigate the precision and recall of suffix queries issued against the 221 bug traces we harvested from the `Rhino` project. We queried OSCILLOSCOPE with 48 unresolved `Rhino` bugs. We found similar bugs for 14 of these bugs, under our experimental procedure. When computing the measures below, we used this result as our oracle for $[\![b]\!]$.

When we do not deeply understand a bug, matching trace suffixes is a natural way to search for bugs, since many bugs cause termination. This insight underlies the suffix query we first introduced in Section 3.2.o For suffixes of length `len`, the suffix query is

```
SELECT bug FROM Rhino WHERE
    SUBSET?(tbug[len], bug[len], distance).
```

Two parameters control suffix comparison: length and edit distance. We conducted two experiments to show how these parameters impact the search for similar bugs. In the first experiment, we fix the suffix length at 50 and increase the allowed Levenshtein distance. In the second experiment, we fix the Levenshtein distance to 10 and vary the suffix length. Table 3.1 depicts the results of the first experiment, Table 3.2 those of the second. We report the data in Table 3.2 in descending suffix length to align the data in both tables in order of increasing match leniency.

In both experiments, we are interested in the number of queries for unresolved bugs that match a resolved bug, as these might help a developer fix the bug. Recall that R is the set of resolved bugs (Section 3.3) and R_b (Equation 3.3.6) is the set of resolved bugs similar to b. In the second column, we report how many unresolved bugs match any resolved bugs. For this purpose, we define $U_R = \{b \in U \mid R_b \neq \emptyset\}$, then, in column three, we report the percentage of unresolved bugs that match at least one resolved bug.

To show that our result sets are accurate, we compute their average size across all unresolved

Distance	$\lvert U_R \rvert$	$\frac{\lvert U_R \rvert}{\lvert U \rvert}$	$\overline{\lvert R_b \rvert}$	$\overline{\frac{\lvert R_b \rvert}{\lvert R \rvert}}$	Average Respr(R,b)	Average Relrec(R,b)	Average F-score(R,b)
0	3	6.3%	0.08	0.05%	0.98	0.75	0.85
10	12	25.0%	0.67	0.39%	0.93	0.94	0.93
20	18	37.5%	1.29	0.75%	0.82	0.96	0.88
30	31	64.6%	2.29	1.32%	0.56	0.98	0.71

Table 3.1: Measures of the utility of our approach as a function of increasing Levenshtein distance and fixed suffix length 50, $\forall b \in U$; U_R is the subset of the unresolved bugs U for which OSCILLOSCOPE finds a similar resolved bug; $\frac{\lvert U_R \rvert}{\lvert U \rvert}$ is the percentage of unresolved bugs that are similar to a resolved bug; $\overline{\lvert R_b \rvert}$ is the average number of resolved bugs returned for each unresolved bug b; since R will vary greatly in size, we report $\overline{\frac{\lvert R_b \rvert}{\lvert R \rvert}}$, the size of each result set as a percentage of the resolved bugs; the meaning of the measures Respr, Relrec and restricted F-score are defined in the text below.

Suffix Length	$\lvert U_R \rvert$	$\frac{\lvert U_R \rvert}{\lvert U \rvert}$	$\overline{\lvert R_b \rvert}$	$\overline{\frac{\lvert R_b \rvert}{\lvert R \rvert}}$	Average Respr(R,b)	Average Relrec(R,b)	Average F-score(R,b)
200	2	4.2%	0.08	0.05%	0.97	0.73	0.83
100	5	10.4%	0.19	0.11%	0.97	0.79	0.87
50	12	25.0%	0.67	0.39%	0.93	0.94	0.93
25	24	50.0%	1.83	1.06%	0.68	0.96	0.80

Table 3.2: Measures of the utility of our approach as a function of decreasing suffix length and fixed distance threshold 10, $\forall b \in U$; for a detailed discussion of the meaning of the first four columns, please refer to the caption of Table 3.1; the remaining columns are defined in the text below.

bugs as

$$\frac{\sum \lvert R_b \rvert}{\lvert U \rvert} = \overline{\lvert R_b \rvert}. \tag{3.4.1}$$

In the fourth column, we report this average, then because R might grow to be very large, we report the average cardinality of $\overline{\lvert R_b \rvert}$ as a percentage of $\lvert R \rvert$. This is the average number of resolved bugs a developer would have to examine for clues he might use to solve an unresolved bug. It is a proxy for developer effort. In this experiment, even the two most lenient matches — Levenshtein

distance 30 and 25 length suffixes — do not burden a developer with a large number of potentially similar bugs. For `Rhino`, the maximum number of bugs returned by OSCILLOSCOPE for a bug is six. The effort to analyze each result set is further mitigated by the fact that OSCILLOSCOPE returns ranked result sets, in order of edit distance.

Next, we report the precision and recall of OSCILLOSCOPE over R, the resolved bugs. When TP denotes true positives, FP denotes false positives, and FN denotes false negatives, recall that

$$\text{precision} = \frac{\text{TP}}{\text{TP}+\text{FP}} \qquad \text{recall} = \frac{\text{TP}}{\text{TP}+\text{FN}}. \tag{3.4.2}$$

In our context, we have

$$\text{precision} = \frac{|[\![b]\!] \cap [b]|}{|[b]|} \qquad \text{recall} = \frac{|[\![b]\!] \cap [b]|}{|[\![b]\!]|}. \tag{3.4.3}$$

To restrict precision to R, we define

$$\text{respr}(X,b) = \begin{cases} 1 & \text{if } [b] \cap X = \emptyset \\ \frac{|([\![b]\!] \cap [b]) \cap X|}{|[b] \cap X|} & \text{otherwise.} \end{cases} \tag{3.4.4}$$

then, in column four, we report the average respr

$$\frac{1}{|U|} \sum_{b \in U} \text{respr}(R,b) \tag{3.4.5}$$

which we manually compute via our experimental procedure. The majority, $\frac{34}{48}$, of the unresolved bugs in our `Rhino` data set are unique. These unique bugs dominate the average respr at lower edit distance and longer suffix length. When distance increases from 10 to 20, and suffix length drops from 50 to 25, the average respr drops dramatically. The reason is that `Rhino`, a JavaScript interpreter, has a standard exit routine. When a program ends without runtime exceptions, `Rhino`

calls functions to free resources. Most the `Rhino` traces end with this sequence which accounts for OSCILLOSCOPE's FPs and decreases the average respr. In general, both Table 3.1 and Table 3.2 show how the greater abstraction comes at the cost of precision.

Our experimental procedure is manual and may overstate recall because we do not accurately account for FNs; accurately accounting for FNs would require examining all traces. Rather than directly report recall (Equation 3.4.3), we over-approximate it with *relative recall*:

$$\text{relrecall}(b) = \begin{cases} 1 & \text{if } [\![b]\!] = \{b\} \\ 1 & \text{if } [\![b]\!] \cap [b] - \{b\} \neq \emptyset \\ 0 & \text{otherwise,} \end{cases} \tag{3.4.6}$$

which measures how often OSCILLOSCOPE returns useful results. Relative recall scores one whenever a bug 1) is unique or 2) has at least one truly similar bug. Because $\forall [b], b \in [\![b]\!] \cap [b]$, we need to test these two cases separately to distinguish between returning only b when b is, in fact, unique in the database from returning b, but failing to return other, similar bugs when they exist. In contrast to precision, relative recall does not penalize the result set for FPs; in contrast to recall, it does not penalize the result set for FNs. In practice, relative recall can be manually verified, since we only need to find a counter-example in $[\![b]\!]$ to falsify the first condition and checking the second condition is restricted by $|[b]|$, whose maximum across our data set is 11.

As with precision, we restrict relative recall

$$\text{relrecall}(X, b) = \begin{cases} 1 & \text{if } [\![b]\!] = \{b\} \\ 1 & \text{if } ([\![b]\!] \cap [b] - \{b\}) \cap X \neq \emptyset \\ 0 & \text{otherwise.} \end{cases} \tag{3.4.7}$$

and, in column five, we report its average,

$$\frac{1}{|U|}\sum_{b\in U}\text{relrecall}(R,b). \tag{3.4.8}$$

By definition two paths are different. At edit distance zero, a bug can only match itself, possibly returning duplicate bug reports. Many paths differ only in which branch they took in a conditional and thus OSCILLOSCOPE can match them even with a small edit distance budget, which accounts for the large rise in relative recall moving from an edit distance budget of 0 to 10. Minor differences accumulate when longer suffixes of two traces are compared. The loss of irrelevant detail accounts for the large rise in relative recall moving from suffix length 100 to 50.

The F-score is the harmonic mean of precision and recall. Here we define it using restricted precision and relative recall:

$$\frac{2\cdot\text{relrecall}(X,b)\cdot\text{respr}(X,b)}{\text{relrecall}(X,b)+\text{respr}(X,b)}. \tag{3.4.9}$$

Column six in both tables reports this measure.

Unique bugs dominate the first Levenshtein measurement. At distance zero, OSCILLOSCOPE returned five bugs in total. Among the 34 unique bugs, OSCILLOSCOPE found a similar bug for only one of them. At suffixes of length 200, OSCILLOSCOPE returned two bugs in total and both are true positives. The distance zero and suffix length 200 queries are more precise, but also more susceptible to FNs, because they return so few bugs. Correctly identified, unique bugs increase relative recall. With the increase of distance or decrease in suffix length, OSCILLOSCOPE finds more potentially similar bugs, at the cost of FPs. At distance 10 and suffixes of length 50, OSCILLOSCOPE found similar bugs for 11 of the non-unique bugs, and returned only one FP for a unique bug.

These tables demonstrate the utility of OSCILLOSCOPE. Its query results often contain re-

solved bugs that may help solve the open, target bug. They are small and precise and therefore unlikely to waste a developer's time. The second most strict measurement, edit distance 10 in Table 3.1 and suffix length 50 in Table 3.2 is the sweet spot where OSCILLOSCOPE is sufficiently permissive to capture interesting bugs while not introducing too many FP.

The six FPs have similar test cases, but their errors are triggered at different program points. For instance, `Rhino-567484` and `352346` have similar test cases that construct and print an `XML` tree. A problem with `XML` construction triggers `567484`, while an error in `XML.toXMLString()` triggers `352346`. Two of FN occurred because the distance threshold was too low. At distance 20, OSCILLOSCOPE matches these two bugs without introducing FPs. Logging calls, using `VMBridge`, separate these two bugs from their peers and required the additional 10 edits to elide. We failed to find similar bugs for `496540` and `496540` because the calls made in `Number.toFixed()` changed extensively enough to disrupt trace similarity. To reduce our FP and FN rates, we plan to investigate object-sensitive dynamic slicing and automatic α-renaming which, given trace alignment, renames symbols, here method names, in order of their appearance. To handle FNs like those caused by the interleaving of new calls such as logging, we intend to evaluate trace embedding, *i.e.* determining whether one trace a subsequence of another, as an additional distance metric.

3.4.2 How Useful are the Results?

In this section, we show how OSCILLOSCOPE can help a programmer fix a bug by identifying bugs similar to that bug. To do so, we issue a suffix query for each bug. The query **Suffix-query**

```
SELECT bug FROM ALL WHERE
    SUBSET?(tbug[50], bug[50], 30)
```

returns a distance-ranked result set for `tbug`. We used trace suffixes of length 50 because OSCILLOSCOPE performed best at this length (Section 3.4.1). We then manually examined the target bug and the bug in the result set with the smallest distance from the target bug, using our experimental

procedure.

As a degenerate case of bug similarity, OSCILLOSCOPE effectively finds duplicate bug reports and helps keep a bug database clean. OSCILLOSCOPE reported 33 clusters of duplicate bug reports, 28 of which the project developers had themselves classified as duplicates. We manually examined the remaining five and confirmed that they are in fact duplicates that had been missed. We have reported these newly-discovered duplicates to the project maintainers. So far, one of them, (`JXPATH-128`, `143`), has been confirmed and closed by the project maintainers. We have yet to receive confirmations on the other four: `Configuration-30` and `256`, `Collections-100` and `128`, `BeanUtils-145` and `151`, and `Rhino-559012` and `573410`. Next, we discuss a selection of interesting, similar bugs returned by **Suffix-query**.

OSCILLOSCOPE is particularly effective at finding API misuse bugs. From the `BeanUtils` project, **Suffix-query** identified as similar `BeanUtils-42`, `117`, `332`, `341`, and `372`, which all incorrectly try to use BeanUtils in conjunction with non-public Java beans. Upon seeing a fix for one of these, even a novice could fix the others. `Configuration-94` and `222` describe bugs that happen when a new file is created without checking for an existing file. `Configuration-94` occurs in `AbstractFileConfiguration.save()` and `222` in the `PropertiesConfiguration` constructor. Their fixes share the same idea: check whether a file exists before creating it. `Lang-118` and `131` violate preconditions of `StringEscapeUtils` `unescapeHtml()}` `Lang-118` found that `StringEscapeUtils` does not handle hex entities (prefixed with '0x'), while `131` concerns empty entities. Both fixes check the input: seeing one, a developer would know to write the other. `Lang-59` (resolved) and `Lang-654` (unresolved) describe the problem that `DataUtils.truncate` does not work with daylight savings time. Studying `Lang-564`, we found its cause was a partial (*i.e.* bad) fix of `Lang-59`. Developers of `Lang` confirmed this finding. `BeanUtils-115` is the problem that properties from a `DynaBean` are not copied to a standard Java bean. In `BeanUtils-119`, `NoSuchMethodException` is thrown when setting value to a property with name "aRa". The root cause for both bugs is the passing of a parameter that violates Java bean naming conventions: when the second character of

a getter/setter method is uppercase, the first must also be. Again, from a fix for either, the fix for the other is immediate.

In addition to identifying bug pairs such as those we just discussed, OSCILLOSCOPE efficiently clusters unresolved bugs. In `Rhino`, OSCILLOSCOPE clustered seven unresolved bugs — `444935`, `443590`, `359651`, `389278`, `543663`, `448443`, and `369860`. These bugs all describe problems with regular expressions. The project developers themselves deemed some of these bugs similar. Problems in processing XML trees causes `Rhino-567484`, `566181` and `566186`. When processing strings, Rhino throws a `StackOverflowException` in both `Rhino-432254` and `567114`.

Although we found interesting, similar bugs comparing only suffix traces, we also learned some of the deficiencies of this strategy. When a bug produces erroneous output and does not throw an exception, the location of the error is usually not at the end of the execution trace. Suffix queries produce FPs on such bugs, especially when they share a prefix because their test cases are similar. The bug `Configuration-6`, improper handling of empty values, and `Configuration-75`, the incorrect deletion of XML subelements, formed one such FP. Suffix comparison can miss similar bugs when a bug occurs in different contexts. The incorrect implementation of `ConfigurationUtils.copy()` caused `Configuration-272`, `283` and `323`. These three bugs differ only in the location of their call to this buggy method. The suffix operator alone failed to detect these bugs as similar, because the context of each call is quite different.

3.4.3 Scalability

Our central hypothesis, that unique bugs are rare against the backdrop of all bugs, means that OSCILLOSCOPE will become ever more useful as its database grows, since it will become ever more likely to find bugs similar to a target bug. This raises performance concern: how much time will it take to process a query against millions of traces. To gain insight into how the size of bug database and Hadoop cluster settings impact the query time, we conducted a two-dimension stress

Database Size	Time to completion (s)					
(million)	1, 4(GB), 4(CPU)	1, 6, 8	2, 10, 12	3, 14, 16	4, 16, 20	5, 144, 38
1	38.2	28.6	23.1	21.1	18.9	14.2
2	72.3	41.3	32.8	24.2	22.5	18.3
3	104.7	53.8	48.4	34.6	28.6	20.2
4	140.7	70.1	58.4	49.4	35.2	25.7
5	171.3	81.2	64.5	53.5	47.9	28.0

Table 3.3: Measures of OSCILLOSCOPE's query processing time against different sizes of databases with different Hadoop cluster settings; The Hadoop cluster setting "*x*, *y*GB *z*CPU" denotes a cluster with *x* nodes, a total of *y* GB of memory, and *z* CPUs.

testing. In the first dimension, we fix the database size and increase Hadoop cluster resources. In the second dimension, we fix the Hadoop cluster and vary the database size. We replicated our 877 traces to create a database with millions of traces. We used the **Suffix-query** as the candidate query in the experiment. For each setting, we issued **Suffix-query** five times and computed the average time to completion. Table 3.3 depicts the results. The first column lists the database size, which ranges from one to five million traces. The rest of the columns show the time to process the query for each each cluster configuration. The Hadoop cluster setting is expressed as the following: "*x*, *y*GB *z*CPU" denotes a cluster with *x* nodes, a total of *y* GB of memory, and *z* CPUs. The settings we used were

Cluster 1 1, 4GB 4CPU

Cluster 2 1, 6GB 8CPU

Cluster 3 2, 10GB 12CPU

Cluster 4 3, 14GB 16CPU

Cluster 5 4, 16GB 20CPU

Cluster 6 5, 144GB 38CPU

Figure 3.4.1 shows that the query processing times grow linearly with the increase of database size for all the settings, the expected result which confirms that all the nodes in the cluster were used. Figure 3.4.1 also shows that query times drop more dramatically for large database sizes than

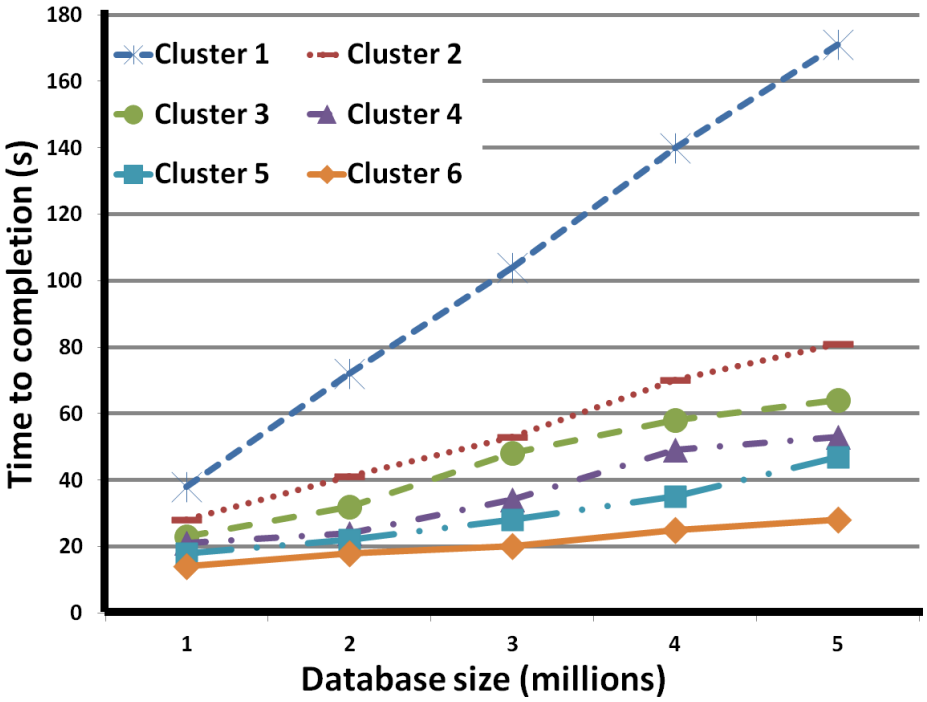


Figure 3.4.1: The effect of database size and Hadoop cluster on query time; The six Hadoop cluster settings vary in number of hosts, total memory and number of CPUs, as described in the text.

for small ones. This is because the performance gain gradually overcomes the network latency as the workload grows. Figure 3.4.1 clearly demonstrates that OSCILLOSCOPE's query processing will take less time as a function of the nodes in its Hadoop cluster.

3.4.4 Execution Trace Search Accuracy

Execution traces accurately capture program behavior, but are expensive to harvest and store. To show this cost is worth paying, we compare the execution trace search accuracy against the accuracy of vector and stack trace searches.

Vector Researchers have proposed a vector space model (VSM) to measure the similarity of execution traces [88]. To compare the VSM metric against OSCILLOSCOPE's use of edit distance over execution traces, we repeated the `Rhino` experiment in Section 3.4.1 using VSM. The VSM

approach transforms execution traces to vectors, then computes their distance[1]. In the experiment, we tuned two parameters, trace length and distance threshold of the VSM algorithm to optimize the respr and relrecall results. We varied the trace lengths using suffixes of length 25, 50, 100, 200, and unbounded. For the distance threshold, we tried $[0.5, 1.0]$ in increments of 0.1. Suffix length 50 and distance threshold 0.9 was the sweet spot of the VSM metric where it achieved 0.85 respr and 0.90 relrecall, for an F-score of 0.87. At suffix of length 50 and edit distance 10, OSCILLOSCOPE outperforms VSM, with 0.93 respr and 0.94 relrecall for and 0.93 F-score. We hypothesize the reason is that the VSM model does not consider the temporal order of method invocations. The search for similar bugs requires two operations, search and insertion. VSM requires the recomputation of the vector for every trace in the database whenever an uploaded trace introduces a new method, but it can amortize this insertion cost across searches, which are relatively efficient because they can make use of vector instructions. In contrast, insertion is free for OSCILLOSCOPE, but search operations require edit distance computations.

Are Execution Traces Necessary? The OSCILLOSCOPE database contains both stack and execution traces. Execution traces are more precise than stack traces, but more expensive to harvest. Stack traces are cheaper to collect, but not always available and less precise. For example, a stack trace is usually not available when a program runs to completion but produces incorrect output; they are less precise because they may not capture the program point at which a bug occurred or when they do, they may capture too little of its calling context. Here, we quantify this difference in the precision to shed light on when the effort to harvest execution traces might be justified.

70 Rhino bugs in our database have stack traces. $33/70$ bugs are similar to at least one other bug, under our experimental procedure. In the experiment, we issued OSCILLOSCOPE queries to search for these similar bug pairs, one restricted to stack traces and the other to execution traces.

[1]Users of OSCILLOSCOPE, who wish to use VSM's distance function, can implement it as a custom distance function.

The stack trace query was

```
SELECT b1, b2 FROM Rhino WHERE SUBSET?(
    PROJ(b1, stack), PROJ(b2, stack)).
```

In the query, `PROJ(b1, stack)` extracts the stack trace of a bug. We ranked each result set in ascending order by distance. We deemed the closest 50 to "match" under OSCILLOSCOPE, since manually examining the 50 pairs was tractable and each of the 70 bugs appears in at least one of these 50 pairs in both result sets. We examined each of these 50 bug pairs to judge whether it is a TP or an FP. Of the remaining pairs, which we deemed non-matches, we examined all the FN.

The stack trace result set contains 26 TP, 24 FP (the 50 that OSCILLOSCOPE matched), and 6 FN (from among the remaining pairs). The fact that most stack traces are similar accounted for nearly half the FPs. The largest distance over all 50 pairs is only 5, compared with 35 for the execution traces. We examined the stack traces of the 6 FNs. The distance between `Rhino-319614` and `370231` is large because the bugs belong to different versions of `Rhino` between which the buggy method was heavily reformulated. Both `Rhino-432254` and `567114` throw stack overflow exceptions whose traces differ greatly in length. Because `Rhino-238699`'s stack trace does not contain the error-triggering point, `Compile.compile()`, it does not match `299539`. The other three FN have distances 8, 14, and 14, so FPs crowd them out of the top 50 pairs.

The execution trace result set contains 41 TP, 9 FP, and 1 FN. False positives appear when the edit distance exceeds 20. The only FN is the pair `Rhino-319614` and `370231`, which is also a FN in the stack trace result set and for the same reason: the relevant methods were extensively rewritten.

The stack trace result set matched similar bugs with 0.52 respr, 0.81 relrecall (See Section 3.4.1) and 0.63 F-score; the execution trace result set achieved 0.82 respr, 0.98 relrecall, and 0.89 F-score. These results make clear that the cost of instrumenting and running executable to collect execution traces can pay off, especially as a fallback strategy when queries against stack traces are inconclusive.

3.4.5 Threats to Validity

We face two threats to the external validity of our results. First, we evaluated OSCILLOSCOPE against bugs collected from the `Rhino` and Apache `Commons` projects, which might not be representative. Second, most of the bug reports we gathered from these projects lack bug-triggering test cases. Without test cases, we were unable to produce traces for almost 70% of the bugs in these projects. Our evaluation therefore rests on the remaining 30%. It may be that those bugs whose reports contain a test case are not representative. These two threats combine to shrink the number of traces over which OSCILLOSCOPE operates.

Nonetheless, our results are promising. We have conjectured that unique bugs are rare, in the limit, as a trace database contains all bug traces. The fact that we have already found useful examples of similar bugs in a small population lends support to our conjecture. Not only is the population small, but it contains only field bugs. In particular, it lacks predeployment bugs, which are resolved during initial development. We contend that these bugs are more likely to manifest common misunderstandings and be more similar than field defects.

Our evaluation is also subject to two threats to its construct validity. First, one cannot know $[\![b]\!]$ in general. We described how we manually approximated $[\![b]\!]$ in our experimental procedure above. Project developers identified 43% of these bugs as similar to another bug. For the remaining 57%, as with any manual process involving non-experts, our assessments may have been incorrect. Second, our database currently contains method, not instruction, granular traces. To the extent to which a method-level trace fails to capture bug semantics, our measurements are inaccurate. Our evaluation data is available at `http://bql.cs.ucdavis.edu`.

3.5 Related Work

This section opens with a discussion of work that, like OSCILLOSCOPE, leverages programming knowledge. Often, one acquires this knowledge to automate debugging, which we discuss

next. We close by discussing work on efficiently analyzing traces.

Reuse of Programmer Knowledge Leveraging past solutions to recurrent problems promises to greatly improve programmer productivity. Most solutions are not recorded. However, the volume of those that are recorded is vast and usually stored as unstructured data on diverse systems, including bug tracking systems, SCM commit messages, and mailing lists. The challenge here is how to support and process queries on this large and unwieldy set of data sources. Each project discussed below principally differs from each other, and OSCILLOSCOPE, in their approach to this problem.

To attack the polluted, semi-structured bug data problem, DebugAdvisor judiciously adds structure, such as constructing bags of terms from documents, and they define a new sort of query, a *fat query*, that unites structured and unstructured query data [2]. In contrast, OSCILLOSCOPE uses execution traces to query bug data. Thus, our approach promises fewer false positives, but may miss relevant bugs, while DebugAdvisor will return larger result sets that may require human processing. Indeed, Ashok *et al.* note "there is much room for improvement in the quality of retrieved results."

When compilation fails, HelpMeOut saves the error message and a snapshot of the source [35]. When a subsequent compilation succeeds, HelpMeOut saves the difference between the failing and succeeding versions. Users can enter compiler errors into HelpMeOut to search for fixes. In contrast, OSCILLOSCOPE handles all bug types and compares execution traces, not error messages. Dimmunix, proposed by Jula *et al.*, prevents the recurrence of deadlock [45]. When a deadlock occurs, Dimmunix snapshots the method invocation sequence along with thread scheduling decisions to form a deadlock signature.

Dimmunix monitors program state. If a program's state is similar to a deadlock signature, Dimmunix either rolls back execution or refuses the lock request. Dimmunix targets deadlock bugs, while OSCILLOSCOPE searches for similar bugs across execution traces of all types of bugs.

Code peers are methods or classes that play similar roles, provide similar functionality, or in-

teract in similar ways. A recurring fix is repeatedly applied, with slight modifications, to several code fragments or revisions. Proposed by Ngueyn *et al.*, FixWizard syntactically finds code peers in source code, then identifies recurring fixes to recommend the application of these fixes to overlooked peers [65]. OSCILLOSCOPE uses traces, which precisely capture bug semantics, to search for similar bugs in a database of existing bugs.

Automated Debugging Detecting duplicate bug reports is a subproblem of finding similar bugs. Runeson *et al.* use natural language processing (NLP) to detect duplicates [78]. Comments in bug reports are often a noisy source of bug semantics, but could complement OSCILLOSCOPE's execution trace-based approach. In a recent work, Wang *et al.* augment the NLP analysis of bug reports with execution information [88]. They record whether or not a method is invoked during an execution into a vector. We compare Wang *et al.*'s approach to ours in Section 3.4.4.

Bug localization and trace explanation aim to find the root cause of a single bug or identify the likely buggy code for manual inspection [5, 16, 27, 28, 39, 41, 44, 48, 57, 58, 90, 95, 96]. OSCILLOSCOPE helps developers fix a bug by retrieving similar bugs and how they were resolved. OSCILLOSCOPE and bug localization complement each other. A root cause discovered by bug localization may lead to the formulation of precise OSCILLOSCOPE queries for similar, resolved bugs; storing only trace subsequence identified by a root cause can also save space in the database.

Efficient Tracing and Analysis Researchers have proposed query languages over traces for monitoring or verifying program properties [69, 70]. Goldsmith *et al.* propose the Program Trace Query Language (PTQL) for specifying and checking a program's runtime behaviors [25]. A PTQL query instruments and runs a program to check whether or not a specified property is satisfied at runtime. Martin *et al.*'s Program Query Language (PQL) defines queries that operate over event sequences of a program to specify and capture its design rules [60]. PQL supports both static and dynamic analyses to check whether a program behaves as specified. Olender and Osterweil propose a parameterized flow analysis for properties over event sequences expressed in Cecil, a constraint language based on quantified regular expressions (QREs) [66]. These three query lan-

guages are designed for analyzing a single, property-specific trace; OSCILLOSCOPE collects raw, unfiltered traces and tackles the bug similarity problem in the form of trace similarity.

3.6 Discussion and Future Work

Debugging is hard work. Programmers pose and reject hypotheses while seeking a bug's root cause. Eventually, they write a fix. To reuse this knowledge about a bug, we must accurately find semantically similar bugs. We have proposed comparing execution traces to this end. We have defined and built OSCILLOSCOPE, a tool for searching for similar bugs, and an open infrastructure — trace collection, a flexible query language BQL, a query engine based on Hadoop, database, and both a web-based and plugin user interface — to support it. BQL allows a user to 1) define bug similarity and 2) use that definition to search for similar bugs. We evaluated OSCILLOSCOPE on a collection of bugs from popular open-source projects. Our results show that OSCILLOSCOPE accurately retrieves relevant bugs: When querying unresolved bugs against resolved bugs in the `Rhino` project it achieves 93% respr, 94% relrecall for an F-score of 0.93 (Section 3.4.1).

OSCILLOSCOPE is an open project. We invite readers to use OSCILLOSCOPE and help us make it more general. Please refer to our website `http://bql.cs.ucdavis.edu` for tutorials and demonstrations.

Chapter 4

IDEPP: Capturing and Exploiting IDE Interactions

Integrated development environments (IDEs) dominate the production and maintenance of software. Developers interact intensively with their IDEs while working. These interactions reflect a developer's thought process and work habits. By capturing and exploiting comprehensive, fine-grained IDE interactions, we can build intelligent IDEs that improve programmer productivity. This next generation of IDEs will incorporate a general framework to capture and exploit IDE interactions, and serve as a basis for an ecosystem of user-aware applications. To this end, we have developed IDE++ on top of the popular Eclipse IDE. We demonstrate comprehensive and granular interaction capture of IDE++ by successfully replaying the interaction process using interaction log. We built four applications upon IDE++ to illustrate 1) the need for capturing comprehensive, fine-grained IDE interactions, and 2) the promise of user-aware applications in IDEs.

4.1 Introduction

Development and maintenance dominate the cost of software. 97% of .NET developers use Microsoft Visual Studio and 73% of Java developers use Eclipse-based IDEs [33]. Thus, increasing the utility and power of IDEs will improve programmer productivity and reduce the cost of software. Interactions between a developer and her IDE capture how the developer writes a piece of code and reflect her thought process and work habits. When we monitor a programmer's IDE interactions, we can look for patterns in the interaction stream that indicate she needs help and provide instant assistance. If we detect that a developer is unsure about which API to use, we can recommend an API and show relevant examples. An IDE can also adapt itself to a programmer's work habits: if it detects that a developer habitually runs testcases after an editing session, it could run relevant tests automatically for her.

We believe the key to building the next generation of IDEs is to transform IDEs from being order-takers into intelligent, user-aware programs that monitor and reason about how their users interact with them, like the Office Assistant in Microsoft Office [40]. We envision establishing an ecosystem of IDE applications and extensions that exploit this awareness to dynamically personalize their interface, to teach their users how to use them more effectively, to help them follow best practices, and to point out features that are likely to be relevant to a developer's current task.

User-aware IDE Applications An ecosystem of behavior-aware IDE applications will benefit the users of IDEs, those who study developers and software processes, and IDE developers. Programmers are often confronted with unexpected, repetitive tasks, like conflict resolution during version control or protocol updates after an API change. A behavior-aware application could identify these cases and suggest macros. User-aware navigation could infer landmarks from the user's behavior, such as frequently returning to a particular class or method, then speed subsequent navigation by jumping to those landmarks. An experienced programmer's interaction log can teach a novice how best to work with an in-house API, refactor a method, fix a bug, or debug

a race condition. When editing an unfamiliar file, a programmer may wish to know which other files other developers who edited that file had opened and which methods had spent the most time on screen, presumably being studied. Interaction logs, suitably sanitized, will allow researchers to investigate questions such as "How do the interaction histories of experienced programmers differ from those of novices?" and "Are there correlations between IDE interactions and bug introduction or cost overruns?". Finally, IDE developers themselves can examine interaction logs to learn which features users actually use to more rapidly improve their IDE's UI.

Capturing IDE Interactions Some behavior-aware applications already exist. The Mylyn Monitor, proposed by Murphy *et al.*, intercepts programmer UI and command interactions to track Eclipse's layout and dynamically reconfigure it to support task-focused workflows [63]. Given its focus, Mylyn Monitor tracks only task-related interactions, such as launches of and switches between views, (*i.e.* GUI windows in Eclipse). Indeed, existing user-aware applications monitor only a very focused and narrow set of IDE interactions: only those necessary for the needs of the application. Moreover, these applications remain the exception not the rule; most IDE interactions are not even captured, let alone recorded. In contrast, we advocate and realize the comprehensive and fine-grained capture of IDE interactions to make it easy to build user-aware applications.

An *IDE interaction* is an action performed by a developer and the IDE's response. IDE-intelligible actions are hardware events, like mouse movement and mouse clicks and keystrokes, that the operating system forwards to the IDE. If an IDE maps a raw sequence of hardware events to an action, the IDE responds by performing the specified action and visually (occasionally auditorily) notifying the user. Thus, hardware events are the atomic constituents of an IDE interaction, comprising either IDE-visible input events or IDE-initiated output events. IDE interaction capture is *fine-grained* the closer it is to the raw hardware events; it is *comprehensive* when it intercepts *every* hardware event visible to an IDE.

Of course, a developer would describe her interaction with an IDE at a fairly high level of abstraction relative to the underlying sequence of raw, low-level, hardware events. For instance,

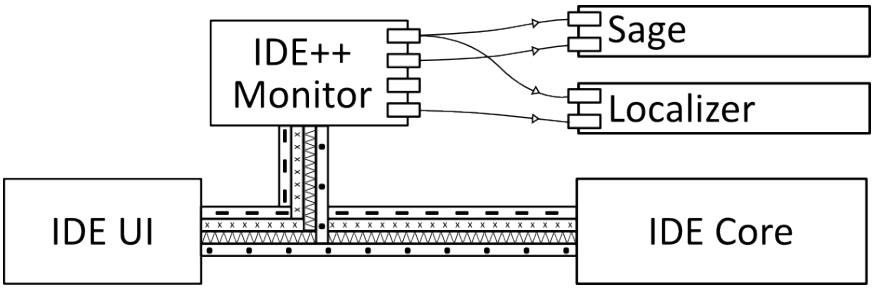


Figure 4.1.1: To support user-aware applications, IDE++ captures and republishes interactions between a user and an IDE.

developers might describe the IDE interactions of a coding session as consisting of a sequence of editing, browsing, testing, and debugging tasks. The appropriate level of abstraction will vary with the development task and as technology changes. It is for this reason that we advocate fine-grained interaction capture: we do wish to preclude any conceivable user-aware IDE application. That said, we capture and can only react to IDE interactions and what we can infer from them. When a huge gap occurs between two user commands, we cannot infer why that gap occurred, nor why a user chose to accomplish a task in a particular order. In short, we do not capture coffee runs.

We have developed IDE++ on top of Eclipse. It consists of an infrastructure that monitors fine-grained programmer interactions in Eclipse and tool support to write custom, user-aware applications. IDE++'s architecture is publish and subscribe, as Figure 4.1.1 shows. IDE++ monitors and extracts IDE interactions, then publishes them to registered applications.

We developed four applications — DevTime, Sage, Proctor and Localizer — to illustrate the promise of an ecosytem of user-aware application built on IDE++ and the necessity of capturing comprehensive, fine-grained interactions. Advocates of the quantified-self movement claim that one can improve oneself through self-study [85]. The DevTime reports descriptive statistics on a developer's activities that facilitate this sort of introspective improvement of one's use of an IDE. For instance, users might discover that conditions under which they, personally, are likely to introduce bugs, perhaps right after they have returned from a coffee break. DevTime summarizes all

of a developer's IDE interactions; its utility rests on comprehensive event capture. Sage teaches novices how to use Eclipse's built-in features to be more productive; the opportunity to use some of these features can only be detected by fine-grained event capture, such as of the keystrokes that comprise manual commenting. Proctor helps to identify which methods programmers should consider testing after an editing session; it collates events from the test UI and editor and demonstrates the need for comprehensive capture. The Localizer uses fine-grained code edits to help programmers determine where their unit test cases to fail.

This research makes the following main contributions:

- We advocate the systematic capture and utilization of IDE interactions as the basis of a new class of user-aware IDE applications;
- We present the IDE++ monitoring infrastructure, built for Eclipse, that comprehensively captures and republishes fine-grained programmer interactions; and
- To demonstrate the promise of the ecosystem of user-aware IDE applications that IDE++ makes possible, we built four applications that require and exploit IDE++'s comprehensive, fine-grained IDE interactions.

The early feedback on IDE++ has been positive and is evidence for its potential for supporting useful customized applications. We have released IDE++ to the Eclipse marketplace. Tutorials and downloads are available at `http://idepp.cs.ucdavis.edu`.

The rest of the chapter is structured as follows. Section 4.2 presents Sage, an application that illustrates the promise of user-aware applications that IDE++ makes possible. In Section 4.3, we specify which interactions we capture and the methodology we use to identify them. Next, Section 4.3 describes the design and implementation of IDE++. In general, it is difficult for one person to flawlessly mimic another. To demonstrate that IDE++'s interaction capture is comprehensive and fine-grained, Section 4.4 shows that one programmer is able to fully replay the IDE

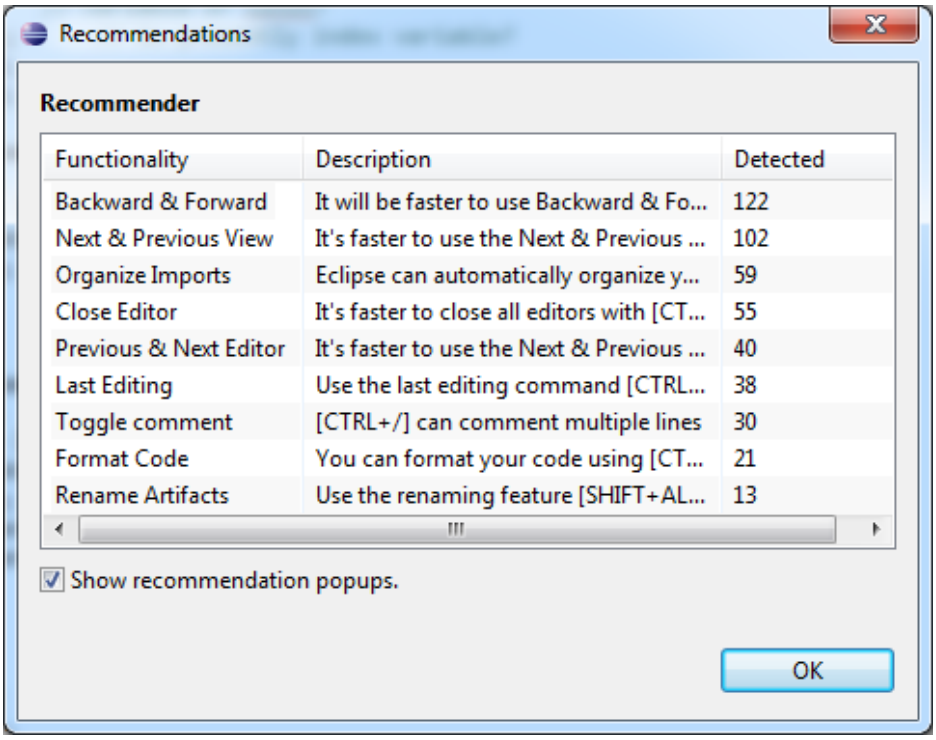


Figure 4.2.1: Sage records how often a programmer could have used a built-in feature, but did not.

session of another programmer, then illustrates the ease of writing applications in IDE++. Finally, Section 4.5 discuss the related work and Section 4.6 concludes.

4.2 Illustrative Example

An IDE is a complicated program, with a lengthy learning curve. Novice IDE users are often unaware of shortcuts. Even programmers who have mastered an IDE may use their IDE inefficiently. For example, a programmer who does not know that Eclipse's Organize-Import feature (hotkey: Shift+Ctrl+O) automatically inserts import statements, must manually type import statements. Although these features are well documented, searching for and reading that documentation often distracts from a developer's current task.

An efficient learning strategy is to have an expert who looks over your shoulder and tells you how to accomplish a task more efficiently. The Sage acts as this angel. It monitors a programmer's

IDE interactions and pops up a tip showing the functionality that would have been faster. For example, Sage pops up a tip teaching the user about the Toggle-Comment feature (hotkey: Ctrl+/), if it detects that a programmer is manually commenting several lines of code. Sage also records the number of times a user could have used a feature but did not. A programmer who uses mouse clicks to switch editors could examine its output in Figure 4.2.1 and see that using Eclipse's Backward and Forward commands might be more efficient for him.

Sage is an example of applying a finite-state machine (FSM) to the interaction stream for self-improvement. To build it, we identified Eclipse features that beginners neglect, manually discovered how to accomplish the task each of these feature speeds up and replaces, then encoded that feature-bypassing sequence of actions into an FSM. While Eclipse is running, Sage continuously feeds the programmer's interactions into each supported feature's automaton. When an automaton reaches a final state, Sage displays the related tip notification. Currently, Sage supports 11 features, shown in Figure 4.2.1. These features are spread across different domains such as editing, refactoring, and browsing; their use saves keystrokes and can improve productivity

4.3 Design and Implementation of IDE++

We integrated IDE++ into Eclipse because it is the dominant IDE for Java programmers. We first describe the methodology we used to realize our goal of comprehensive and fine-grained interaction capture in IDE++. We then describe IDE++'s architecture, how its clients can subscribe to its events and extend it to capture events from new plug-ins.

4.3.1 Methodology

Programmers interact with IDEs through GUI windows called views. Every interaction is tied to a view. Components group views with the same purpose. As an example, Table 4.1 lists all the components, and their constituent views, in Eclipse's default configuration. Different views

Component	Views
General	Bookmarks, Console, Error Log, Markers, Navigator, Outline, Problems, Progress, Project Explorer, Properties, Search, Tasks, Templates
Development	Code editor, Call Hierarchy, Declaration, Hierarchy, Javadoc, Package Explorer
Browsing	Members, Packages, Projects, Types
Debug	Breakpoints, Debug, Display, Expressions, Memory, Modules, Registers, Variables
Testing	Unit testing

Table 4.1: The components and their views of the default configuration of Eclipse.

define disjoint sets of interactions. Programmers can edit code in the editor view, not a search view. A programmer interacts with an IDE in one view at one time. He can switch to another view by performing a special interaction view switch such as opening the JUnit view or issuing the Previous-View command. Sometimes view switching is implicit: running a test case after editing a program switches the view from editing to testing.

To ensure IDE++ captures comprehensive and fine-grained interactions, we first sought to identify all Eclipse views, then, for each view, all interactions defined by that view. In both cases, we systematically studied the Eclipse GUI, its documentation, and, finally, its source code. In particular, when seeking to identify all the interactions in a view, we reasoned from first principles, asking ourselves what interactions a particular view *must* have in order to achieve the goal for which it was designed. We began our search for Eclipse interaction with the GUI components in Eclipse's default configuration.

Take Code editor view as an example. The most fine-grained interactions we capture are file buffer changes: inserting or removing one character in a file. Some fine-grained interactions might construct a high level interaction. For example, code changes also reflect language-specific semantics, such as AST changes. In Java language, a bunch of code inserting interactions might

form up adding a field in a class. Our model also capture this high level of changes as interactions.

Eclipse's JUnit framework provides support for running tests inside of Eclipse. After systematically analyzing it as described above, we determined it would be best to extend their `TestRunListener` class. This notifies us of the user starting and ending a JUnit test session, along with notifications each time a test case starts and finishes. Components supported in Eclipse is a moving target. Currently, we support all the views in default configuration for Java development.

Of course, we cannot know the set of IDE interactions precisely, both because of differences among IDEs and because, as technological advances, IDEs will acquire new features that define new actions and views. For this reason, we have taken pains to make sure it easy to extend IDE++ to new plug-in and view, as described in Section 4.4.2.

4.3.2 The Architecture of IDE++

At its core, IDE++ realizes the publish and subscribe model on two levels. In the context of Eclipse, its host IDE, IDE++ is itself a subscriber that listens for both incoming, IDE-visible events and IDE responses to those events. It is with respect to these events that IDE++ strives to be comprehensive and fine-grained. IDE++ then republishes these events to the ecosystem of user-aware applications that it enables; from the perspective of its clients, IDE++ is a publisher to which they subscribe. IDE++ collects IDE interactions in three ways: For view switching, it extends Mylyn Monitor; to capture edits, it directly instruments the Eclipse editor; for the rest, it augments Eclipse's default interaction collection facilities.

The Mylyn Monitor, given its focus on supporting task-oriented workflows, monitors all view-switch interactions (exposed in `IPartListener` in Eclipse) and we used their listener `MonitorUi` directly. However, for user selection interactions, it provides only one event for all kinds of selections which is not comprehensive. To get fine-grained interactions, IDE++ refactored the listener by splitting single method into sets of methods. As an example, IDE++ refactored `handleWorkbenchPartSelectio`

into `structuredSelection`, the method `textSelection`, and `otherSelection` to specialize them for different selection contents. When capturing an interaction, IDE++ also extracts associated information, such as the name of an opened view.

The most challenging interactions to capture are editing interactions. The Eclipse JDT plug-in manages Java editor events in a decentralized way: it provides listeners in different modules to capture mixed of fine- and coarse-grained change notifications. For example, `IElementChangedListener` publishes AST changes and `IDocumentListener` notifies file buffer changes. Besides, the support is not comprehensive. For example, it fails to provide refactoring notifications. We aimed to provide more consistent, meaningful and fine-grained change types including the low-level editor buffer change and high-level language semantic changes such as refactoring. To capture fine grained text editing interactions, we implemented two listeners. First, we implemented the `IFileBufferListener` interface to signal the opening or closing of a text file buffer. Once we know a text file buffer has been opened, we get its backing document and attach an `IDocumentListener` to capture `DocumentEvent` instances. A document event contains the offset, the length of the change, and the text of the change if it is an insertion. Eclipse's support for attaching a listener to refactoring interactions is poor. For instance, when a user performs a copy or rename refactor operation, a refactoring event is fired to `RefactoringExecutionEvent` listeners. However, this event does not give enough information to fully determine the changes that will be performed. To solve this problem, we built modules that participate in the refactoring process and determine the changes that will be done from the information passed to them.

Eclipse provides hooks to allow developers to register interaction monitors; users need to implement listeners exposed. For example, to monitor how users change a class in Java development, IDE++ need to implement `IElementChangedListener`, which shows when a change was made to a class such as adding a new field. However, some listeners do not support comprehensive interactions. The `IElementChangedListener` fails to notify all class changes, for example, renaming a field. For some events, the monitoring process in Eclipse is centralized: it provides one listener

```
 1  2012-03-28 18:16:56,090|main|35|95|p
 2  2012-03-28 18:16:56,650|main|35|96|u
 3  2012-03-28 18:16:56,762|main|35|97|b
 4  2012-03-28 18:17:01,002|main|36|97|1
 5  2012-03-28 18:17:01,711|main|36|96|1
 6  2012-03-28 18:17:01,961|main|35|96|r
 7  2012-03-28 18:17:01,995|main|35|97|i
 8  2012-03-28 18:17:02,137|main|35|98|v
 9  2012-03-28 18:17:02,153|main|35|99|a
10  2012-03-28 18:17:02,165|main|35|100|t
11  2012-03-28 18:17:02,170|main|35|101|e
12  2012-03-28 18:17:04,172|main|28|Save
```

Figure 4.3.1: Sample interaction log recorded by IDE++

for all the events in different views. `CommandMonitor` is an example; it monitors all the command events from all the views. To be comprehensive, we refactored listener to have one listener for each view. For example, we have `DebugCommandListener` for all commands in Debug view: Resume, Suspend, Step-into, and etc.

IDE++ extensively instruments Eclipse to intercept interactions. Generally, instrumenting programs vastly slows them down, and often imposes an unacceptable performance penalty. The instrumentation IDE++ adds, however, is unique in that it is confined to IO with a human. Humans are glacially slow compared to computers; relative to human reaction time which average 190ms [92], IDE++'s overhead is imperceptible.

4.3.3 Interaction History

The syntax of a single interaction captured by IDE++ contains four fields: time stamp, thread id, type, and content. The type field, recorded as an integer, identifies an interaction, such as a view-selection or a edit. Currently, IDE++ supports 44 kinds of interactions, documented at `http://idepp.cs.ucdavis.edu`. Some interactions have associated content. For instance, an editing interaction includes the characters that have been typed; these characters are stored in the

content field.

Figure 4.3.1 shows an example of interactions captured by IDE++. Type 35 denotes keystroke and type 36 denotes a Backspace keystroke. The numbers (95-101) after the type information of the interactions record the offset of the edits in the file. The characters that were typed follow the offset. The last interaction (type 28) tells that the user performed "Save" command. The sample log records the following actions performed by a user: He begins to create a public field. Then he decides to change it to private. So he removes "ub", types "rivate" and clicks "Save".

IDE++ interaction logs grow quickly. To save space and minimize performance overhead, we store only required information and use buffered writers to write into the log files in order to minimize performance overhead. Recall that, as note above, the instrumentation IDE++ adds to capture IDE interaction is on the very slow path to a human. We exploit this fact to perform online event-filtering that would otherwise be prohibitively expensive and relegated to postprocessing. For this reason, IDE++ guards all its republishing events with tests that check whether any listener is been registered for that event.

4.3.4 Subscribing to IDE++ Events

IDE++ supports both online and offline analysis of interaction information. The online analysis is a prerequisite for building "smart" IDEs that know what a programmer is doing and offer live assistance. It will form the basis of an ecosystem of user-aware IDE applications. The offline analysis allows retrospective analysis a programmer's interactions. The IDE++ infrastructure enables any plug-in to subscribe to its interaction information. Internally, IDE++ sets up monitors when Eclipse launches and receives events while Eclipse remains open. It provides a set of listener APIs to which applications can subscribe to receive interactions.

Currently, IDE++ offers six listener interfaces. The first listener, `DebugBreakpointListener`, captures breakpoint interactions; `DocumentChangesListener` captures changes in documents; `JavaLaunchListene`

```
 1 notifyAdded( IJavaElement element );
 2 notifyCodeChanged( IJavaElement element );
 3 notifyCopied(
 4         IJavaElement element, IJavaElement from,
 5         IJavaElement to
 6 );
 7 notifyMoved( IJavaElement from, IJavaElement to );
 8 notifyRemoved( IJavaElement element );
 9 notifyRenamed( IJavaElement from, IJavaElement to );
10 notifySignatureChanged( IJavaElement element );
11 notifySuperTypesChanged( IType type );
```

Figure 4.3.2: API methods in `JavaModelListener`.

captures program launch information; `JavaModelListener` captures editing interactions; `JUnitListener` captures JUnit interactions; and, the final listener, `UserActivityListener`, captures UI and command interactions. Figure 4.3.2 shows the API methods in `JavaModelListener`. Developers only need to implement listeners that capture the interactions they want. To lower the burden on developers, IDE++ provides adapters with do-nothing implementations of the listener interfaces. These adapters allow developers to to focus on their application's logic instead of littering their code with irrelevant methods.

To persist a programmer's interactions, IDE++ leverages its own framework of listeners: The log file is produced by a meta-listener that implements and registers with all of IDE++'s listeners. This listener then echos incoming events into the log file. There are two main concerns regarding log files: 1) log files might become very large and 2) programmers do not want sensitive information such as source code and author information to leak to unauthorized applications. The measures IDE++ takes to handle log size are addressed above in Section 4.3.3. IDE++ is designed to support local IDE plug-ins. Thus, IDE++ handles privacy concerns in the same way Microsoft Excel does — *viz.* share nothing by default and instead leave the management of the log files to the discretion of the user. In addition, users can choose to hash the concrete information such as the source code edits to prevent it from being leaked.

To help developers build plug-ins, we have published documentation, tutorials and examples showing how to use the IDE++ infrastructure at `http://idepp.cs.ucdavis.edu`.

4.3.5 Extending IDE++

The set of IDE interactions is a moving target. New plug-ins will introduce new interactions. IDE++ has an open design that allows integration of new interactions easily. Integrating a new interaction requires two steps: 1) extending the subscriber to monitor the new plug-in to get change notifications and 2) adding a new listener in the publisher side to allow client to retrieve interactions. We illustrate the two steps using the EGit plug-in as an example.

To subscribe to change notifications from a new plug-in, we need to find out the listeners it provides. As an example, EGit provides an `IndexChangedListener` that is notified when the Git index changes. While Eclipse is running, IDE++ will receive notifications from EGit and then publish them to the the IDE++ listeners.

The remaining step is to allow other applications to receive the new interactions from the new publisher side. First, we need to parse the notification object to retrieve or determine useful information. In this example, we decide to retrieve the Repository object from the notification. We then call the relevant `notifyIndexChanged` method on the IDE++ listeners and pass the repository as a parameter. Other applications can now implement the listener and register it with IDE++ to get access to this piece of the interaction stream.

The subscriber and publisher architecture makes monitoring and exploiting interactions straightforward. Often, IDE++ needs only to subscribe to a plug-in's interactions then republish them to other applications. When this is not the case, it should be easy for a developer who is familiar with the plug-in export its interaction to IDE++. Finally, extending a new plug-in is a one time task that opens the door to IDE++'s ecosystem of user-aware applications.

Task	Min.	Edit		Browse		Test		Debug		Total	
		IDE++	Mylyn	IDE++	Mylyn	IDE++	Mylyn	IDE++	Mylyn	IDE++	Mylyn
RPNCalculator	32	1,367	12	98	98	25	2	39	28	1,529	140
String	23	888	10	142	142	12	1	9	1	1,051	154
Repeat-Until	44	3,211	26	144	144	64	3	23	7	3,442	180
Array	94	8,233	47	246	246	81	3	24	13	8,584	309
Boundary	25	720	8	165	165	40	2	11	3	936	178
Every	40	3,020	25	228	228	5	1	3	1	3,256	255

Table 4.2: A comparison of the interactions captured by IDE++ and Mylyn Monitor.

4.4 Evaluation

Our evaluation objective is two-fold: to demonstrate that IDE++ comprehensively captures fine-grained IDE interactions, and to show the promise of that information as the basis of an ecosystem of user-aware IDE applications.

4.4.1 Comprehensiveness and Granularity

To be the basis of a vibrant ecosystem of user-aware applications, IDE++ must effectively realize the goal of comprehensive and fine-grained interactions capture. Here, we present two experiments that measure the degree to which we succeeded. The first experiment shows that we are able to fully replay a nontrivial sequences of IDE interaction from IDE++'s interaction history. The second experiment quantifies IDE++'s event capture, using Mylyn Monitor as a baseline.

IDE Interaction Replay In this experiment, IDE++ records the actions of Programmer A as he performs some programming tasks. Then we show that, given the same initial environment as A had and using A's interaction log, programmer B can redo exactly what A did and produce the same output.

Table 4.2 lists the six programming tasks we used in this experiment. The RPNCalculator task requires a programmer to write a reverse polish notation calculator and provide JUnit testcases to

Task	Participant	Time (min)
RPNCalculator	Student A	21
String	Student A	15
Repeat-Until	Student B	33
Array	Student B	61
Boundary	Student C	21
Every	Student C	33

Table 4.3: Time used for participants to replay the interactions for each task.

ensure correctness. An undergraduate course in our department assigned programming tasks that involved adding support for syntactic constructions to a pedagogical language E by translating them into C. The five constructs were Strings, Repeat-Until, Arrays, Array Boundary check, and Every, a loop construct similar to a foreach. One of the authors completed these tasks while IDE++ recorded his interactions. The second column records the time he used to finish each task.

IDE++'s raw output is not easy for human to parse, so we postprocessed it to separate user actions from Eclipse's responses and to map file offsets into a line number and column. Figure 4.4.1 shows postprocessed output. For the replay experiment, the participant simply follows the user actions in the log.

We invited three students to participate in the experiment. All of them had moderate coding experience and were familiar with the Eclipse IDE. Each participant performed the replay experiment for two of the six assignments in Table 4.2. Table 4.3 shows the time taken to replay the interactions for each programming task. We diffed their source code files against the target files and confirmed that they match. To make sure the entire process was replayed, we also compared the logs produced by the participants with the author's logs and confirmed that the interactions recorded are the same with the exception of the time stamps.

Mylyn Monitor Comparison Like IDE++, Murphy *et al.*'s Mylyn Monitor captures IDE interactions; its focus is capturing those interactions needed to understand and support task-oriented

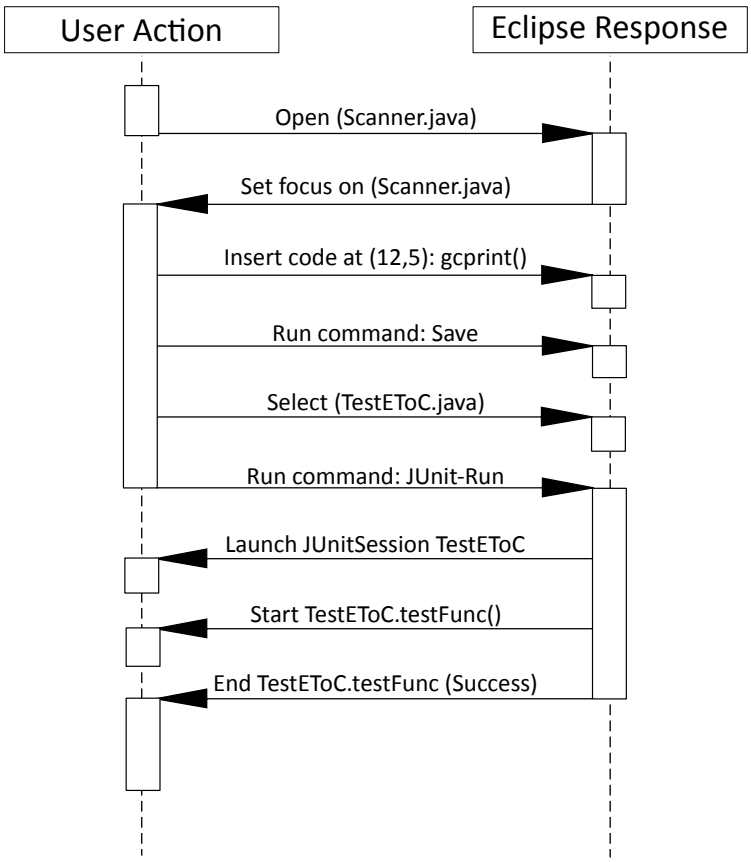


Figure 4.4.1: Sample natural language interaction sequence diagram.

workflows. In contrast, IDE++ seeks to be a general purpose framework for a new class of user-aware IDE applications. Although the two projects differ in focus, Mylyn Monitor is a mature, well-engineered project. Thus, we use it as a baseline against which to understand the scope and detail of IDE++'s interaction capture.

To enable the comparison of interaction logs between IDE++ and the Mylyn Monitor, we normalized the both project's interactions into atomic actions performed by a user, such as a mouse click. We categorized the actions into four categories, editing, browsing, testing, and debugging, and compared the number of actions recorded for each group.

Table 4.2 shows the number of actions captured by both monitors for each category. It is clear that IDE++ captures far more editing interactions than the Mylyn Monitor does. This is because

IDE++ captures all of the fine-grained interactions including cursor movement, keystrokes, and related commands, while the Mylyn Monitor records only coarse-grained file change events and commands. Browsing actions include selecting structured content and switching views; the Mylyn Monitor was designed for this task and IDE++ builds on and inherits from it, as the data makes clear. For the testing category, the Mylyn Monitor records only that the Run-Test command was performed, while IDE++ also includes which testcases have been run and their results (success or failure). For debugging, IDE++ records when a user enables, disables, or changes a breakpoint, the debugging commands a user uses, such as Step-Into and Step-Over, which variables he has inspected while his program was paused, and the stack frames he selected. The Mylyn Monitor records only the commands run and that a variable or stack frame was selected, but no data about it. Table 4.2 clearly shows that IDE++ successfully captures a wide range of fine-grained interactions and provides data from the Mylyn Monitor as baseline for comparison.

4.4.2 User-aware IDE Applications

To demonstrate the necessity of comprehensive and fine-grained interaction information and the promise of an ecosystem of user-aware applications built on IDE++, we introduce, in addition to Sage (introduced in Section 4.2), three IDE++ applications: DevTime, Proctor, and Localizer. These applications help programmers edit, test, and debug. We show the source code of a simple yet meaningful application to illustrate how easy it is to write an application using IDE++.

DevTime After finishing a task, a programmer may want to review what he did to track a project's progress or file a daily working report. Typically, he would review those changes in his version control system (VCS). However, VCS history is a coarse record of what he actually did: it does not reflect the time he spent browsing code or running regression tests. If he made several changes in a single location to the source in his editor, VCS can capture only those changes actually committed to its history. DevTime has two reports: a summary of task performed by

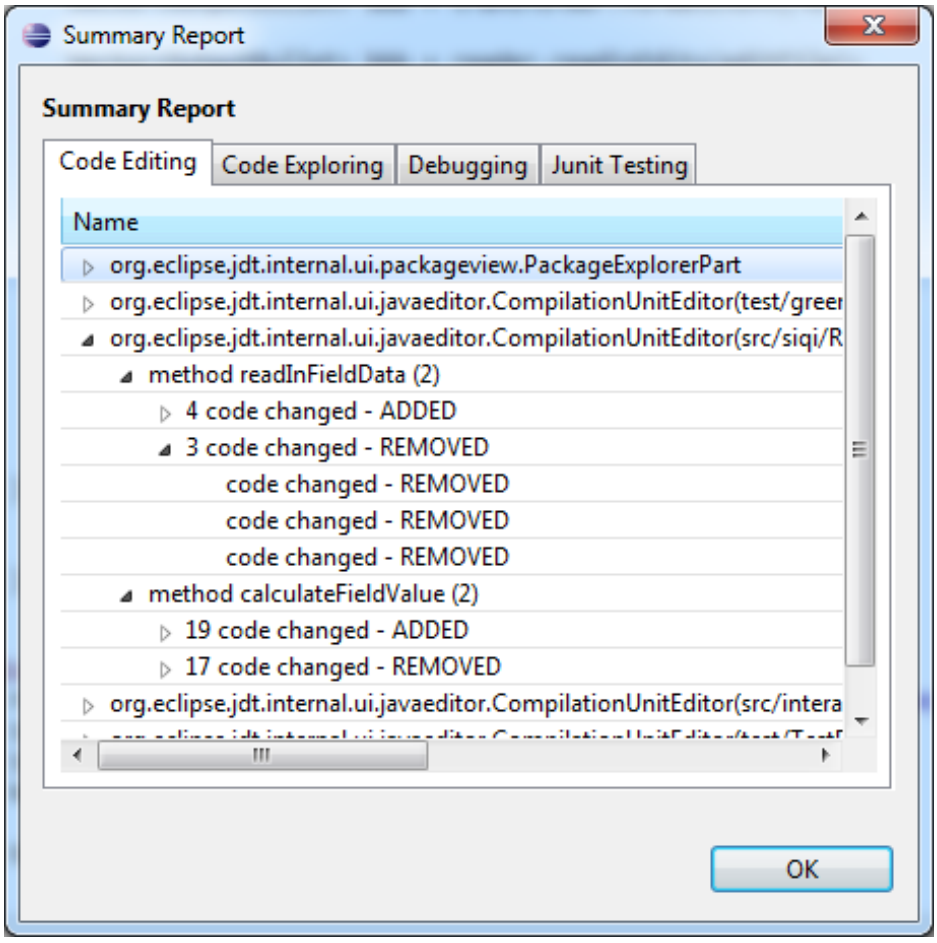


Figure 4.4.2: The Summary Report application shows a programmer how he has interacted with Eclipse.

category, shown in Figure 4.4.2, and a timeline visualization in Figure 4.4.3.

Proctor A good software engineering practice is to test a method while the method and changes made to it are still fresh in a programmer's mind. The longer the gap between editing and testing, the harder it is for a programmer to find and fix a bug he introduced. After a series of edits, a programmer implicitly builds a testing plan. If he knows which methods have been edited and tested recently, it will be much easier for him to remember what needs to be tested.

Proctor helps programmers build testing plans by tracking which methods have been edited and tested recently. It monitors the editing interactions to get the list of methods that have been

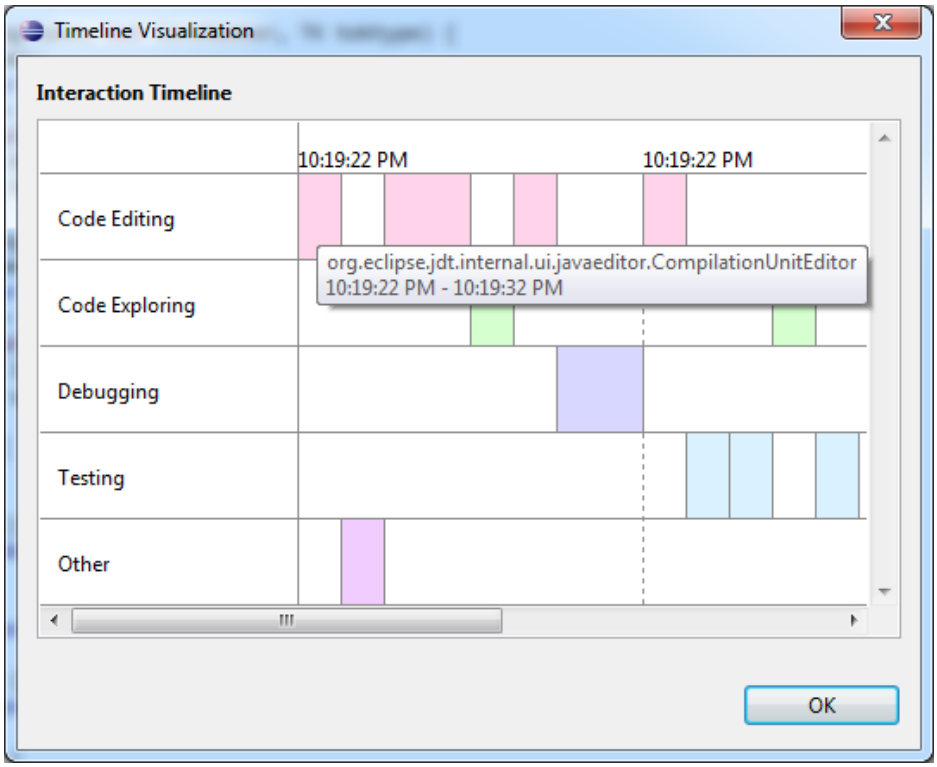


Figure 4.4.3: IDE++ draws a timeline of a user's interactions for the most recent session.

changed and JUnit test interactions to get the list of run test cases. By scanning the source code of the test methods, the Proctor knows which methods have been tested and whether they passed the test or not. It organizes the edited methods into three categories and presents them to the programmer: methods that have not been tested yet, methods that have been tested and passed, and methods that have been tested but failed. Figure 4.4.4 shows example output.

Localizer Bug localization is an active research area. Researchers have proposed various approaches to localizing bugs: statistical models, code history, program slicing, etc. [44, 46, 57, 59, 95]. Many techniques apply sophisticated analysis to an entire program.

The IDE++ Localizer introduces a new approach: localizing bugs by searching the recent editing history. Programmers run regression test cases routinely to ensure that recent edits have not adversely affected existing functionality. When a test case fails, it is likely that a recent edit

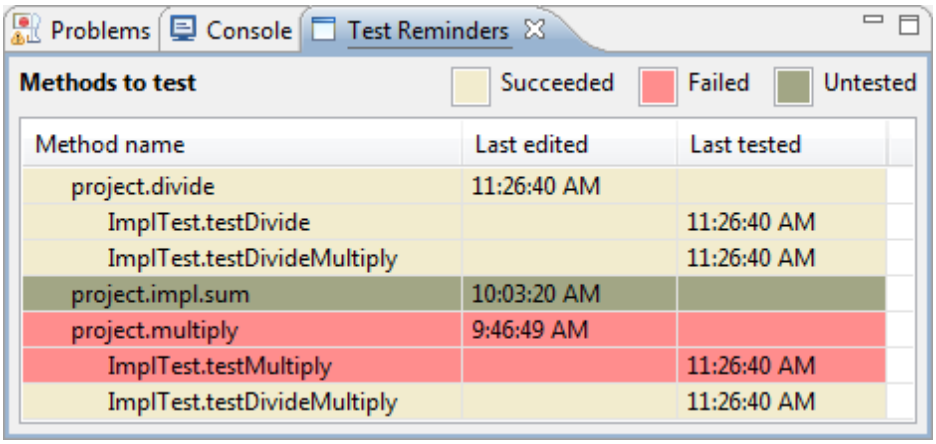


Figure 4.4.4: Proctor tracks editing and testing interactions to remind the user which methods have not been tested yet.

```
public int sum(int a, int b) {
//  return a + b;
  return a + b + 3;
}

public void sayHello() {
//  System.out.println("Hello world");
  System.out.println(
          "Goodbye cruel world"
  );
}
```

Figure 4.4.5: Programmer's code change.

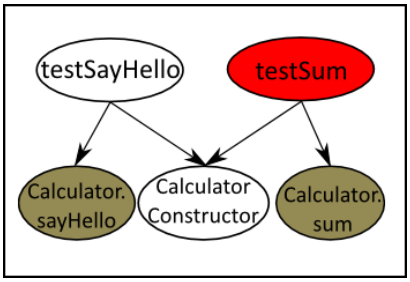


Figure 4.4.6: Call Graph of test case methods.

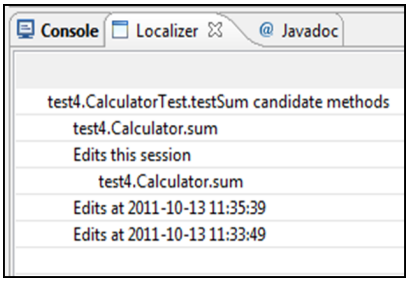


Figure 4.4.7: The Localizer result.

Figure 4.4.8: How Localizer works: (a) A programmer changes both `sayHello` and `sum` (marked gray in (b)). (b) `testSum` fails when the test cases were run (marked red). (c) Localizer suggests that changes in `sum` might have caused the failure.

caused the failure. Using this heuristic and the program's call graph, the IDE++ Localizer lists recently edited methods that might have caused a JUnit test failure. Since this approach requires only recent editing history, it is light-weight and provides live feedback.

Consider the example shown in Figure 4.4.8. A programmer changed both the `sum` and `sayHello` methods in `Calculator` during the current session. When he ran the test cases, `testSum` failed. Although both `sum` and `sayHello` were changed, since the call graph of `testSum` shows that only `sum` affects it, the Localizer tells the programmer that `sum` is the candidate method that

```
1  class RunningAverage {
2          int count;
3          double average;
4          RunningAverage(double a) {
5                  count = 1; average = a;
6          }
7  }
8  JavaLaunchListener l = new JavaLaunchListener() {
9    private Map<IType, Long> launched =
10     new HashMap<IType, Long>();
11   private Map<IType, Long> runningAverages =
12     new HashMap<IType, RunningAverage>();
13   public void programLaunched( IType mainType ) {
14     launched.put(
15       mainType, System.currentTimeMillis()
16     );
17   }
18   public void programTerminated( IType mainType ) {
19     Long started = launched.remove( mainType );
20     if (started != null) {
21       long runTime =
22         System.currentTimeMillis() - started;
23       RunningAverage rAve = runningAverages.get(mainType);
24       if (rAve == null) {
25         runningAverages.put(
26           mainType, new RunningAverage( runTime );
27         );
28       }
29       else {
30         rAve.average *= rAve.count;
31         rAve.average += runTime;
32         rAve.average /= ++rAve.count;
33       }
34     }
35   }
36 };
37 IDEPPPlugin.addListener(l);
```

Figure 4.4.9: Instrumenting program launch interactions.

caused the failure.

Localizer illustrates the use of a particular kind of interaction; it monitors only the JUnit launch and editing interactions, and is triggered by a a test failure. First, it builds the set of the methods called by the test method. Then it extracts the set of methods edited during the recent sessions from IDE++'s interaction history. The intersection of these two sets forms the set of candidates. Finally, Localizer presents these candidates to the programmer. Since the edits triggering the error might not have occurred in the most recent session, Localizer can search the edit history of previous sessions. By default, Localizer searches the last three sessions. The programmer can override this default.

	LOC	
Application	**Interaction**	**Total**
DevTime	20	972
Sage	278	2,123
Proctor	69	889
Localizer	130	874

Table 4.4: Line of code (LOC) of the four applications.

Writing IDE++ Applications

Table 4.4 displays the lines of code (LOC) of the four applications we built. The third column lists the total LOC; second column shows the LOC related to receiving and processing IDE interactions from IDE++. DevTime retrieves interactions directly from IDE++ log files, so its LOC for interactions is only 20. Because Sage monitors every available interaction and parses the arguments to get information, it requires the most logic to handle interactions. Additionally, it contains many different automata that check for different patterns, explaining why it is much larger than the other three plug-ins. Comparing the second and third columns, we see that developers need to write very little code to retrieve and utilize interaction information from IDE++.

We use a simple, albeit, real example to illustrate how easy it is to build an IDE++ application. Assume a developer wants to track the average running time of the programs run during an Eclipse session. For this application, the developers needs only implement and register `JavaLaunchListener` to intercept program launch interactions. In Figure 4.4.9, the developer puts the bulk of the time-averaging logic in `handleTerminated` and stores the results in `launched`. Although it is quite short, the code demonstrates a complete use of the IDE++ infrastructure.

4.5 Related Work

Many IDEs, Eclipse among them, support rudimentary user monitoring [84]. They provide hooks that allow developers to implement their own listeners. However, the support for capturing interactions is ad hoc and cumbersome. For example, Eclipse only partially captures UI interactions, since it does not capture mouse actions; when a programmer issues a command, Eclipse does not report whether the programmer typed a hotkey or clicked a button in the tool bar or a menu. In contrast, IDE++ comprehensively captures and republishes IDE interactions in a standard, easily parsed format.

Code evolution dominates the software life cycle. Developers use a version control system (VCS) to track code evolution. To support collaborative development, VCS allows users to write and commit code to a shared repository. IDE++ also tracks code evolution, but at a finer granularity: we capture every edit interaction, as the changes in an buffer between two idle periods of at least one second. When a developer is editing, he might make several changes at the same location in the source before committing it. IDE++ captures all of the edits while a VCS captures only the difference between commits. By capturing these granular edits instead of file saves, IDE++ comes closer to capturing a developer's thought process. For example, a tricky problem might cause a developer to navigate back and forth between files, make and unmake a change, before he reaching a decision.

Murphy *et al.*'s Mylyn Monitor was an important step in the capture of IDE interactions [63]. Indeed, many recent IDE applications, which we discuss next, depend on the Mylyn Monitor. IDE++ continues the Mylyn Monitor's pioneering work along three dimensions — comprehensiveness, ease-of-use and granularity. IDE++ seeks to intercept all IDE interactions, as identified by the phases of the software life cycle. The set of IDE interactions is a moving target, so at any instance in time, especially when a new plugin gains traction, IDE++ will fall short of this goal. Thus, IDE++ has been designed to make it easy for programmers to extend it to new classes of

interactions.

Robbes and Lanza proposed Spyware an IDE monitor that captures fine-grained editing interactions [73]. Like IDE++, Spyware is a framework on which to build applications. Unlike IDE++, Spyware exclusively intercepts edits, ignoring other IDE interactions. Vakilian *et al.*'s CodingSpectator and CodingTracker aim to capturing low-level code refactoring changes [86]. Yoon *et al.* present Fluorite that captures low-level editing interactions [94]. The above work all focus on a subset of interactions, while our work advocate systematic monitoring all kinds of interactions.

4.5.1 Applications

Program comprehension is an important part of the software engineering process. Researchers have applied interaction information to aid program comprehension. Fritz *et al.* proposed a model using interactions captured by the Mylyn Monitor to judge a programmer's knowledge of code [23]. Kersten and Murphy use the Mylyn Monitor to produce a recommender for the next method to edit using their degree of interest (DOI) measure, which is derived from a database of interaction traces [47]. Guzzi *et al.* proposed a new type of interaction, collective code bookmark to summarize source code to help programmers understand software artifacts [31]. Guzzi *et al.* also presented a micro-blogging technique: group a series of interactions and attach a message to describe the interactions to enhance program comprehension [32]. Ko *et al.* studied the relationship between interactive aspects of IDEs and program understanding [52]. IDE++ can complement these applications by providing more information in the form of finer-grained interactions and additional classes of interactions, such a debugging interactions. For example, the sequence of debugging commands a programmer issues might indicate his degree of knowledge of some code.

A large body of work on in-program assistative agents for general purpose applications exists. The Lumiere project, which culminated in the Office Assistant in Microsoft Office, is one notable

example [40]. Lumiere's focus is Bayesian learning, but also internally abstracts the event stream to explicitly represent repetition and inter-event gaps, with which we intend to experiment. A more recent example is Ekstrand *et al.*'s work, where the researchers are interested in marrying free-form text query to in-program context sensitivity [19].IDE++ can be seen as the specialization of this line of work to the IDE domain; We believe its ecosystem of user-aware applications will quickly grow to encompass those that apply Machine Learning to its stream of interactions.

Brun *et al.* proposed speculative analysis that leverages idle multicores to anticipate what a user may next wish to do, such as compile or run JUnit, and kick off these tasks in the background, shifting them to the foreground if the guess is correct [8]. For instance, they speculatively apply Eclipse's quick fix tips in the background, then tell users which ones worked. For version control interactions, they speculatively merge a developer's current branch in the background to report how many conflicts would arise [9]. IDE++ allows the extension of speculation to other programmer interactions. For example, by monitoring editing interactions, the IDE could run relevant test methods in the background, and notify the programmer about failures.

Researchers have also employed interaction history for prediction. Robbes *et al.* used Mylyn Monitor interaction logs to improve program change prediction [75]. Robbes and Lanza used edit history to improve IDE code completion [74]. Lee *et al.* described a set of micro interaction metrics, such as how much time a programmer spends in one file and how many selection operations he makes to predict bugs [54]. Purandare *et al.* present a general framework for optimizing the monitoring of loops [71]. IDE++'s fine-grained interaction information provides more data as input to predictors. As an example, Lee's bug prediction model could include editing interactions: A file that has been changed many times is likely to contain bugs. IDE++ allows the construction of prediction models for new classes of development activities, such as running a test.

4.6 Discussion and Future Work

The interactions between programmers and their IDEs contain valuable information, much of which currently goes to waste. Systematically recorded into an easy-to-use format, these interactions will usher in new, highly personalized, user-aware applications with the potential to improve programming productivity. In this chapter, we have introduced IDE++, an IDE interaction monitor, and four applications — DevTime, Sage, Proctor, and Localizer — to demonstrate the promise and utility of the new ecosystem of applications IDE++ creates. We plan to extend IDE++ to support more interactions, such as support for compiler warning and collaboration plug-ins such as EGit and Subclipse. We have published these applications as well as the IDE++ infrastructure onto the Eclipse Marketplace. We welcome users to try it.

Programmer interaction histories are good candidates for using data mining techniques to discover previously unknown patterns. Experienced programmers' interactions facilitate knowledge reuse and provide new educational opportunities. Of course, sharing the interaction information raises privacy concerns. We intend apply existing techniques, such as CQual [22], taint analysis [64], and sanitization [89], to IDE++ to protect contributors.

Our monitoring infrastructure is an open framework. We welcome other developers to build applications upon it. Tutorials, documentation, tool downloads, and updates are available at `http://idepp.cs.ucdavis.edu`.

Chapter 5

CONCLUSION

This dissertation has proposed three research efforts that aim to ease the debugging process. Each of these efforts focuses on a different stage of debugging. Oscilloscope leverages knowledge from existing bugs to help developers fix new bugs, and targets the first stage: understanding and fixing a bug. Once a bug is fixed, Fixation helps developers verify the bug fix to avoid the bad fix problem. It targets the stage of testing and validating bug fixes. IDEPP, an infrastructure for monitoring and capturing developer activities, helps understand how developers debug and build applications to aid debugging. It is applicable throughout the software development life cycle. This chapter concludes the dissertation by briefly discussing interesting future work on OSCILLOSCOPE and IDEPP.

OSCILLOSCOPE is an open infastructure, and its database will grow from our activities and the contributions from others. With more and more bugs in its database, it will become a valuable resource for empirical software engineering. We will use it to study questions such as "Can we quantify the precision-scale trade-off of varying trace granularity?", "What proportion of bugs can only be captured at statement granularity?", and "Which parts of the system do few buggy traces traverse?" We also plan to add the traces of correct executions to OSCILLOSCOPE's database, which currently contains only buggy executions and compare them. We intend to ex-

plore adding call contexts to produce execution trees. Because OSCILLOSCOPE is a general and flexible framework, adding this support requires only modifying the instrumentation component and defining a distance measure over execution trees. In short, we intend to enhance and further experiment with our framework to gain additional insights into "What makes bugs similar?"

IDEPP serves as a basis for an ecosystem of user-aware applications. The future work of IDEPP can be categorized into two directions: applications and empirical research. First, it enables developers to build intelligent IDEs and applications to improve programming productivity. We will continue building more useful applications based on IDEPP and improving the infrastructure to enable other developers to contribute. Second, the systematic and fine-grained interactions can be used to study how programmers work and how projects evolve. As our interaction database grows, we will use it to study questions such as "How do the interaction histories of experienced programmers differ from those of novices?", "Do programmers from the same project share common interaction patterns?", and "Does interaction history correlate with measures such as program complexity and bug density?"

Bibliography

[1] J. Anvik, L. Hiew, and G. C. Murphy. Who should fix this bug? In *Proceedings of the 28th international conference on Software engineering*, pages 361–370, New York, NY, USA, 2006. ACM.

[2] B. Ashok, J. Joy, H. Liang, S. K. Rajamani, G. Srinivasa, and V. Vangala. Debugadvisor: a recommender system for debugging. In *Proceedings of the the 7th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*, pages 373–382, New York, NY, USA, 2009. ACM.

[3] T. Ball and S. K. Rajamani. The slam project: debugging system software via static analysis. In *Proceedings of the 29th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 1–3, New York, NY, USA, 2002. ACM.

[4] C. Barrett and C. Tinelli. Cvc3. In *Proceedings of the 19th international conference on Computer aided verification*, pages 298–302, Berlin, Heidelberg, 2007. Springer-Verlag.

[5] I. Beer, S. Ben-David, H. Chockler, A. Orni, and R. Trefler. Explaining counterexamples using causality. In *Proceedings of the 21st International Conference on Computer Aided Verification*, pages 94–108, Berlin, Heidelberg, 2009. Springer-Verlag.

[6] N. Bettenburg, S. Just, A. Schröter, C. Weiss, R. Premraj, and T. Zimmermann. What makes

a good bug report? In *Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of software engineering*, pages 308–318, New York, NY, USA, 2008. ACM.

[7] D. Beyer, A. J. Chlipala, T. A. Henzinger, R. Jhala, and R. Majumdar. Generating tests from counterexamples. In *Proceedings of the 26th International Conference on Software Engineering*, pages 326–335, Washington, DC, USA, 2004. IEEE Computer Society.

[8] Y. Brun, R. Holmes, M. D. Ernst, and D. Notkin. Speculative analysis: exploring future development states of software. In *Proceedings of the FSE/SDP workshop on Future of software engineering research*, pages 59–64, New York, NY, USA, 2010. ACM.

[9] Y. Brun, R. Holmes, M. D. Ernst, and D. Notkin. Proactive detection of collaboration conflicts. In *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*, pages 168–178, New York, NY, USA, 2011. ACM.

[10] Bugzilla. `http://www.bugzilla.org/`.

[11] S. Chandra, S. J. Fink, and M. Sridharan. Snugglebug: a powerful approach to weakest preconditions. In *Proceedings of the 2009 ACM SIGPLAN conference on Programming language design and implementation*, pages 363–374, New York, NY, USA, 2009. ACM.

[12] J. S. Collofello and S. N. Woodfield. Evaluating the effectiveness of reliability-assurance techniques. *J. Syst. Softw.*, 9(3):191–195, Mar. 1989.

[13] C. Csallner and Y. Smaragdakis. Check 'n' crash: combining static checking and testing. In *Proceedings of the 27th international conference on Software engineering*, pages 422–431, New York, NY, USA, 2005. ACM.

[14] C. Csallner, Y. Smaragdakis, and T. Xie. Dsd-crasher: A hybrid analysis tool for bug finding. *ACM Trans. Softw. Eng. Methodol.*, 17(2):8:1–8:37, May 2008.

[15] M. d'Amorim, S. Lauterburg, and D. Marinov. Delta execution for efficient state-space exploration of object-oriented programs. In *Proceedings of the 2007 international symposium on Software testing and analysis*, pages 50–60, New York, NY, USA, 2007. ACM.

[16] P. Dhoolia, S. Mani, V. S. Sinha, and S. Sinha. Debugging model-transformation failures using dynamic tainting. In *Proceedings of the 24th European conference on Object-oriented programming*, pages 26–51, Berlin, Heidelberg, 2010. Springer-Verlag.

[17] E. W. Dijkstra. *A Discipline of Programming*. October 1976.

[18] J. Dolby, M. Vaziri, and F. Tip. Finding bugs efficiently with a sat solver. In *Proceedings of the the 6th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*, pages 195–204, New York, NY, USA, 2007. ACM.

[19] M. Ekstrand, W. Li, T. Grossman, J. Matejka, and G. Fitzmaurice. Searching for software learning resources using application context. In *Proceedings of the 24th annual ACM symposium on User interface software and technology*, UIST '11, pages 195–204, New York, NY, USA, 2011. ACM.

[20] 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 *Proceedings of the eighteenth ACM symposium on Operating systems principles*, pages 57–72, New York, NY, USA, 2001. ACM.

[21] C. Flanagan, K. R. M. Leino, M. Lillibridge, G. Nelson, J. B. Saxe, and R. Stata. Extended static checking for java. In *Proceedings of the ACM SIGPLAN 2002 Conference on Programming language design and implementation*, pages 234–245, New York, NY, USA, 2002. ACM.

[22] J. S. Foster, M. Fähndrich, and A. Aiken. A theory of type qualifiers. In *Proceedings of*

the ACM SIGPLAN 1999 conference on Programming language design and implementation, pages 192–203, New York, NY, USA, 1999. ACM.

[23] T. Fritz, G. C. Murphy, and E. Hill. Does a programmer's activity indicate knowledge of code? In *Proceedings of the the 6th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*, pages 341–350, New York, NY, USA, 2007. ACM.

[24] P. Godefroid, N. Klarlund, and K. Sen. Dart: directed automated random testing. In *Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*, pages 213–223, New York, NY, USA, 2005. ACM.

[25] S. F. Goldsmith, R. O'Callahan, and A. Aiken. Relational queries over program traces. In *Proceedings of the 20th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 385–402, New York, NY, USA, 2005. ACM.

[26] T. L. Graves, M. J. Harrold, J.-M. Kim, A. Porter, and G. Rothermel. An empirical study of regression test selection techniques. *ACM Trans. Softw. Eng. Methodol.*, 10(2):184–208, Apr. 2001.

[27] A. Groce, S. Chaki, D. Kroening, and O. Strichman. Error explanation with distance metrics. *Int. J. Softw. Tools Technol. Transf.*, 8(3):229–247, June 2006.

[28] A. Groce and W. Visser. What went wrong: explaining counterexamples. In *Proceedings of the 10th international conference on Model checking software*, pages 121–136, Berlin, Heidelberg, 2003. Springer-Verlag.

[29] Z. Gu, E. T. Barr, and Z. Su. Bql: capturing and reusing debugging knowledge. In *Proceed-*

ings of the 33rd International Conference on Software Engineering, pages 1001–1003, New York, NY, USA, 2011. ACM.

[30] D. Gusfield. *Algorithms on Strings, Trees, and Sequences: Computer Science and Computational Biology*. Cambridge University Press, 1997.

[31] A. Guzzi, L. Hattori, M. Lanza, M. Pinzger, and A. v. Deursen. Collective code bookmarks for program comprehension. In *Proceedings of the 2011 IEEE 19th International Conference on Program Comprehension*, pages 101–110, Washington, DC, USA, 2011. IEEE Computer Society.

[32] A. Guzzi, M. Pinzger, and A. van Deursen. Combining micro-blogging and ide interactions to support developers in their quests. In *Proceedings of the 2010 IEEE International Conference on Software Maintenance*, pages 1–5, Washington, DC, USA, 2010. IEEE Computer Society.

[33] J. S. Hammond. IDE usage trends, 2008.

[34] M. J. Harrold, J. A. Jones, T. Li, D. Liang, A. Orso, M. Pennings, S. Sinha, S. A. Spoon, and A. Gujarathi. Regression test selection for java software. In *Proceedings of the 16th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 312–326, New York, NY, USA, 2001. ACM.

[35] B. Hartmann, D. MacDougall, J. Brandt, and S. R. Klemmer. What would other programmers do: suggesting solutions to error messages. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, pages 1019–1028, New York, NY, USA, 2010. ACM.

[36] H. He and N. Gupta. Automated debugging using path-based weakest preconditions. In *Fundamental Approaches to Software Engineering, Springer, LNCS*, pages 267–280, 2004.

[37] T. A. Henzinger, R. Jhala, R. Majumdar, and G. Sutre. Lazy abstraction. In *Proceedings of the 29th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 58–70, New York, NY, USA, 2002. ACM.

[38] T. A. Henzinger, R. Jhala, R. Majumdar, and G. Sutre. Software verification with blast. In *Proceedings of the 10th international conference on Model checking software*, pages 235–239, Berlin, Heidelberg, 2003. Springer-Verlag.

[39] K. J. Hoffman, P. Eugster, and S. Jagannathan. Semantics-aware trace analysis. In *Proceedings of the 2009 ACM SIGPLAN conference on Programming language design and implementation*, pages 453–464, New York, NY, USA, 2009. ACM.

[40] E. Horvitz, J. Breese, D. Heckerman, D. Hovel, and K. Rommelse. The lumire project: Bayesian user modeling for inferring the goals and needs of software users. In *Proceedings of the Fourteenth Conference on Uncertainty in Artificial Intelligence*, pages 256–265. Morgan Kaufmann, 1998.

[41] E. W. Host and B. M. Ostvold. Debugging method names. In *Proceedings of the 23rd European Conference on ECOOP 2009 — Object-Oriented Programming*, pages 294–317, Berlin, Heidelberg, 2009. Springer-Verlag.

[42] IBM. T.J. Watson libraries for analysis. `http://wala.sf.net`.

[43] IDC. A Telecom and Networks market intelligence firm. `http://www.idc.com`.

[44] J. A. Jones, M. J. Harrold, and J. Stasko. Visualization of test information to assist fault localization. In *Proceedings of the 24th International Conference on Software Engineering*, pages 467–477, New York, NY, USA, 2002. ACM.

[45] H. Jula, D. Tralamazza, C. Zamfir, and G. Candea. Deadlock immunity: enabling systems to defend against deadlocks. In *Proceedings of the 8th USENIX conference on Operating*

systems design and implementation, pages 295–308, Berkeley, CA, USA, 2008. USENIX Association.

[46] M. Kamkar, N. Shahmehri, and P. Fritzson. Bug localization by algorithmic debugging and program slicing. In *Proceedings of the 2nd International Workshop on Programming Language Implementation and Logic Programming*, pages 60–74, London, UK, UK, 1990. Springer-Verlag.

[47] M. Kersten and G. C. Murphy. Using task context to improve programmer productivity. In *Proceedings of the 14th ACM SIGSOFT international symposium on Foundations of software engineering*, pages 1–11, New York, NY, USA, 2006. ACM.

[48] S. Kim, K. Pan, and E. E. J. Whitehead, Jr. Memories of bug fixes. In *Proceedings of the 14th ACM SIGSOFT international symposium on Foundations of software engineering*, pages 35–45, New York, NY, USA, 2006. ACM.

[49] S. Kim, T. Zimmermann, E. J. Whitehead Jr., and A. Zeller. Predicting faults from cached history. In *Proceedings of the 29th international conference on Software Engineering*, pages 489–498, Washington, DC, USA, 2007. IEEE Computer Society.

[50] J. C. King. Symbolic execution and program testing. *Commun. ACM*, 19(7):385–394, July 1976.

[51] A. J. Ko and B. A. Myers. A framework and methodology for studying the causes of software errors in programming systems. *J. Vis. Lang. Comput.*, 16(1-2):41–84, Feb. 2005.

[52] A. J. Ko, B. A. Myers, M. J. Coblenz, and H. H. Aung. An exploratory study of how developers seek, relate, and collect relevant information during software maintenance tasks. *IEEE Trans. Softw. Eng.*, 32(12):971–987, Dec. 2006.

[53] J. R. Larus. Whole program paths. In *Proceedings of the ACM SIGPLAN 1999 conference on Programming language design and implementation*, pages 259–269, New York, NY, USA, 1999. ACM.

[54] T. Lee, J. Nam, D. Han, S. Kim, and H. P. In. Micro interaction metrics for defect prediction. In *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*, pages 311–321, New York, NY, USA, 2011. ACM.

[55] V. Levenshtein. *Binary codes capable of correcting deletions, insertions and reversals (in Russian)*. 1965.

[56] B. Liblit. *Cooperative bug isolation*. Springer-Verlag, Berlin, Heidelberg, 2007.

[57] B. Liblit, A. Aiken, A. Zheng, and M. Jordan. Bug isolation via remote program sampling. In *Proceedings of the 2003 ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2003.

[58] B. Liblit, M. Naik, A. X. Zheng, A. Aiken, and M. I. Jordan. Scalable statistical bug isolation. In *Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*, pages 15–26, New York, NY, USA, 2005. ACM.

[59] C. Liu, X. Yan, L. Fei, J. Han, and S. P. Midkiff. Sober: statistical model-based bug localization. In *Proceedings of the 10th European software engineering conference held jointly with 13th ACM SIGSOFT international symposium on Foundations of software engineering*, pages 286–295, New York, NY, USA, 2005. ACM.

[60] M. Martin, B. Livshits, and M. S. Lam. Finding application errors and security flaws using pql: a program query language. In *Proceedings of the 20th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 365–383, New York, NY, USA, 2005. ACM.

[61] T. J. McCabe. A Complexity Measure. *IEEE Transactions on Software Engineering*, 2(4), 1976.

[62] S. McCamant and M. D. Ernst. Predicting problems caused by component upgrades. In *Proceedings of the 9th European software engineering conference held jointly with 11th ACM SIGSOFT international symposium on Foundations of software engineering*, pages 287–296, New York, NY, USA, 2003. ACM.

[63] G. C. Murphy, M. Kersten, and L. Findlater. How are java software developers using the eclipse ide? *IEEE Softw.*, 23(4):76–83, July 2006.

[64] J. Newsome and D. X. Song. Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software. In *NDSS*, 2005.

[65] T. T. Nguyen, H. A. Nguyen, N. H. Pham, J. Al-Kofahi, and T. N. Nguyen. Recurring bug fixes in object-oriented programs. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 1*, pages 315–324, New York, NY, USA, 2010. ACM.

[66] K. M. Olender and L. J. Osterweil. Cecil: A sequencing constraint language for automatic static analysis generation. *IEEE Trans. Softw. Eng.*, 16(3):268–280, Mar. 1990.

[67] A. Orso, N. Shi, and M. J. Harrold. Scaling regression testing to large software systems. pages 241–251, 2004.

[68] S. Person, M. B. Dwyer, S. Elbaum, and C. S. Păsăreanu. Differential symbolic execution. In *Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of software engineering*, pages 226–237, New York, NY, USA, 2008. ACM.

[69] G. Pothier and E. Tanter. Summarized trace indexing and querying for scalable back-in-time

debugging. In *Proceedings of the 25th European conference on Object-oriented programming*, pages 558–582, Berlin, Heidelberg, 2011. Springer-Verlag.

[70] G. Pothier, E. Tanter, and J. Piquer. Scalable omniscient debugging. In *Proceedings of the 22nd annual ACM SIGPLAN conference on Object-oriented programming systems and applications*, pages 535–552, New York, NY, USA, 2007. ACM.

[71] R. Purandare, M. B. Dwyer, and S. Elbaum. Monitor optimization via stutter-equivalent loop transformation. In *Proceedings of the ACM international conference on Object oriented programming systems languages and applications*, pages 270–285, New York, NY, USA, 2010. ACM.

[72] C. S. Păsăreanu, P. C. Mehlitz, D. H. Bushnell, K. Gundy-Burlet, M. Lowry, S. Person, and M. Pape. Combining unit-level symbolic execution and system-level concrete execution for testing nasa software. In *Proceedings of the 2008 international symposium on Software testing and analysis*, pages 15–26, New York, NY, USA, 2008. ACM.

[73] R. Robbes and M. Lanza. Spyware: a change-aware development toolset. In *Proceedings of the 30th international conference on Software engineering*, pages 847–850, New York, NY, USA, 2008. ACM.

[74] R. Robbes and M. Lanza. Improving code completion with program history. volume 17, pages 181–212, Hingham, MA, USA, June 2010. Kluwer Academic Publishers.

[75] R. Romain, P. Damien, and L. Michele. Replaying IDE interactions to evaluate and improve change prediction approaches. In *MSR*, 2010.

[76] G. Rothermel and M. J. Harrold. Analyzing regression test selection techniques. *IEEE Trans. Softw. Eng.*, 22(8):529–551, Aug. 1996.

[77] G. Rothermel and M. J. Harrold. A safe, efficient regression test selection technique. *ACM Trans. Softw. Eng. Methodol.*, 6(2):173–210, Apr. 1997.

[78] P. Runeson, M. Alexandersson, and O. Nyholm. Detection of duplicate defect reports using natural language processing. In *Proceedings of the 29th international conference on Software Engineering*, pages 499–510, Washington, DC, USA, 2007. IEEE Computer Society.

[79] B. G. Ryder and F. Tip. Change impact analysis for object-oriented programs. In *Proceedings of the 2001 ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering*, pages 46–53, New York, NY, USA, 2001. ACM.

[80] R. Santelices, P. K. Chittimalli, T. Apiwattanapong, A. Orso, and M. J. Harrold. Test-suite augmentation for evolving software. In *Proceedings of the 2008 23rd IEEE/ACM International Conference on Automated Software Engineering*, pages 218–227, Washington, DC, USA, 2008. IEEE Computer Society.

[81] A. Schröter, N. Bettenburg, and R. Premraj. Do stack traces help developers fix bugs? In *Proceedings of the 7th IEEE Working Conference on Mining Software Repositories*, 2010.

[82] K. Sen, D. Marinov, and G. Agha. Cute: a concolic unit testing engine for c. In *Proceedings of the 10th European software engineering conference held jointly with 13th ACM SIGSOFT international symposium on Foundations of software engineering*, pages 263–272, New York, NY, USA, 2005. ACM.

[83] J. Śliwerski, T. Zimmermann, and A. Zeller. When do changes induce fixes? In *Proceedings of the 2005 international workshop on Mining software repositories*, pages 1–5, New York, NY, USA, 2005. ACM.

[84] The Eclipse Foundation. Eclipse instrumentation framework.

http://dev.eclipse.org/viewcvs/viewvc.cgi/platform-ui-home/instrumentation/inde x .html?revision=1.12.

[85] The Economist. The quantified self: Counting every moment. *The Economist Magazine*, 2012.

[86] M. Vakilian, N. Chen, S. Negara, B. A. Rajkumar, R. Zilouchian Moghaddam, and R. E. Johnson. The need for richer refactoring usage data. In *Proceedings of the 3rd ACM SIGPLAN workshop on Evaluation and usability of programming languages and tools*, pages 31–38, New York, NY, USA, 2011. ACM.

[87] W. Visser, C. S. Păsăreanu, and S. Khurshid. Test input generation with Java PathFinder. In *ISSTA '04: Proceedings of the 2004 ACM SIGSOFT international symposium on Software testing and analysis*, 2004.

[88] X. Wang, L. Zhang, T. Xie, J. Anvik, and J. Sun. An approach to detecting duplicate bug reports using natural language and execution information. In *Proceedings of the 30th international conference on Software engineering*, pages 461–470, New York, NY, USA, 2008. ACM.

[89] G. Wassermann and Z. Su. Sound and precise analysis of web applications for injection vulnerabilities. In *Proceedings of the 2007 ACM SIGPLAN conference on Programming language design and implementation*, pages 32–41, New York, NY, USA, 2007. ACM.

[90] W. Weimer, T. Nguyen, C. Le Goues, and S. Forrest. Automatically finding patches using genetic programming. In *Proceedings of the 31st International Conference on Software Engineering*, pages 364–374, Washington, DC, USA, 2009. IEEE Computer Society.

[91] C. Weiss, R. Premraj, T. Zimmermann, and A. Zeller. Predicting effort to fix software bugs. In *Proceedings of the 9th Workshop Software Reengineering*, May 2007.

[92] Wikipedia. Mental chronometry. http://en.wikipedia.org/wiki/Mental_chronometry, 2012.

[93] J. Wloka, E. Hoest, and B. G. Ryder. Tool support for change-centric test development. *IEEE Software*, 99(PrePrints), 2009.

[94] Y. Yoon and B. A. Myers. Capturing and analyzing low-level events from the code editor. In *Proceedings of the 3rd ACM SIGPLAN workshop on Evaluation and usability of programming languages and tools*, pages 25–30, New York, NY, USA, 2011. ACM.

[95] A. Zeller. Yesterday, my program worked. today, it does not. why? In *Proceedings of the 7th European software engineering conference held jointly with the 7th ACM SIGSOFT international symposium on Foundations of software engineering*, pages 253–267, London, UK, UK, 1999. Springer-Verlag.

[96] X. Zhang, N. Gupta, and R. Gupta. Locating faults through automated predicate switching. In *Proceedings of the 28th international conference on Software engineering*, ICSE '06, pages 272–281, New York, NY, USA, 2006. ACM.