Injecting Mechanical Faults to Localize Developer Faults for Evolving Software

Lingming Zhang[1] Lu Zhang[2] Sarfraz Khurshid[1]

[1]Department of Electrical and Computer Engineering, University of Texas, Austin, 78712, USA
Email: zhanglm@utexas.edu, khurshid@ece.utexas.edu
[2]Key Laboratory of High Confidence Software Technologies (Peking University), MoE, Beijing, 100871, China
Email: zhanglu@sei.pku.edu.cn

Abstract

This paper presents a novel methodology for localizing faults in code as it evolves. Our insight is that the essence of failure-inducing edits made by the developer can be captured using mechanical program transformations (e.g., mutation changes). Based on the insight, we present the FIFL framework, which uses both the spectrum information of edits (obtained using the existing FAULTTRACER approach) as well as the potential impacts of edits (simulated by mutation changes) to achieve more accurate fault localization. We evaluate FIFL on real-world repositories of nine Java projects ranging from 5.7KLoC to 88.8KLoC. The experimental results show that FIFL is able to outperform the state-of-the-art FAULTTRACER technique for localizing failure-inducing program edits significantly. For example, all 19 FIFL strategies that use both the spectrum information and simulated impact information for each edit outperform the existing FAULTTRACER approach statistically at the significance level of 0.01. In addition, FIFL with its default settings outperforms FAULTTRACER by 2.33% to 86.26% on 16 of the 26 studied version pairs, and is only inferior than FAULTTRACER on one version pair.

Categories and Subject Descriptors D2.5 [*Software Engineering*]: Testing and Debugging

General Terms Algorithms, Experimentation

Keywords Software Evolution; Regression Testing; Fault Localization; Mutation Testing

OOPSLA '13, October 29–31, 2013, Indianapolis, Indiana, USA.

http://dx.doi.org/10.1145/2509136.2509551

1. Introduction

The problem of *fault localization*, i.e., identifying the locations of faulty lines of code, remains challenging, often requiring much manual effort. This paper presents a novel solution to this problem in the context of code that evolves. Our insight is that the essence of failure-inducing edits made by the developer can be simulated using mechanical program transformations. Specifically, some transformations are likely to share the same locations with failure-inducing edits if those transformations transform the old program version (i.e., the version right before the faults were introduced) to have similar test pass/fail results as the new version with real developer edits.

To simulate developer edits, we use program transformations based on *mutation testing* [4, 9, 12, 14, 18, 20, 34, 42, 43, 56], which is a methodology originally designed for measuring test-suite quality based on injected faults. Mutation testing generates variants (termed *mutants*) for the original program under test using mechanical transformation rules (termed *mutation operators*). Each mutant is the same with the original program except for the mutated statement. A mutant is termed *killed* by a test suite if some test from the suite produces different results on the mutant and the original program. Empirical studies have shown that test suites that kill a high percentage of mutants are likely to reveal more faults and mutation testing is often viewed as the strongest test criteria [3, 11]. It has been used to evaluate the quality of existing test suites [9, 34, 42, 47] and to generate test suites with high quality [8, 12, 18, 35, 56].

To our knowledge, mutation changes have not been utilized to simulate developer edits to achieve precise fault localization. The existing approaches for fault localization during software evolution mainly use sole coverage information of developer edits. *Change impact analysis* [40] is a well-known methodology for determining *affecting changes*, i.e., a subset of program edits that might have caused the test failure, based on edit coverage information in regression testing [19, 40, 57]. It has been shown that the number of af-

fecting changes for each failed test can still be large [57]. Therefore, various techniques have been proposed to localize failure-inducing changes more precisely [39, 45, 57]. The recently developed FAULTTRACER approach [57] introduces spectrum-based fault localization [1, 23, 28, 52] to localization of failure-inducing edits; experimental results show that FAULTTRACER significantly outperforms Ren et al.'s ranking heuristic [39] based on test call graph structures. However, FAULTTRACER still suffers from lack of accuracy, because the *spectrum* information (i.e., the edits that are mainly executed by failed tests are considered more suspicious) is still based on only coverage information and the suspicious edits may not be responsible for test failures. For instance, some edits are ranked at the top just because they are *accidentally* executed by failed tests.

A straightforward idea to refine the fault localization results is to automatically apply various subsets of program edits according to their ranking to localize failure-inducing edits more precisely. However, there are three basic reasons that make this idea impractical. First, program edits might have complex compilation dependences between them, which does not allow them to be applied independently. Second, iteratively applying various combination of edits may cost extra test execution time. Third, iteratively applying program edits does not work for concurrent programs, since some faulty edits may be missed just because they accidentally passed the test suite once. As a result, existing techniques for localizing failure-inducing edits usually recommend *manually* applying and inspecting edits after ranking them [5, 39, 57].

Although real program edits by developers and mechanical mutation changes by mutation testing are both changes to the original program, they are traditionally treated as two separate dimensions. This paper unifies these two dimensions of changes. We use both the spectra of edits (obtained using FAULTTRACER) as well as the potential impacts of edits (simulated by mutation changes) to achieve more accurate fault localization.

This paper presents our methodology of fault injection for fault localization (FIFL) and our framework that embodies it for achieving more precise fault localization during software evolution based on mutation testing. To localize failure-inducing edits, FIFL first utilizes the mutation testing results on the old version[1] and gets the test execution results for each mutant. Second, following FAULTTRACER, FIFL uses spectrum-based techniques [1, 23, 28, 52] to calculate the suspiciousness of program edits between the old and new versions. Third, FIFL builds a mapping between program edits with mutants of the old version that can potentially simulate the corresponding edits based on a set of inference rules. Fourth, FIFL determines the impacts of mutation changes by calculating the similarity between test execution results (Pass/Fail) of the mutants for the old version with the test execution results of the new version, and treats the similarity as the suspiciousness of mutants. Finally, for every program edit, FIFL refines its suspiciousness based on the suspiciousness of its mapped mutants, because those mutants can simulate the potential impact of the edit.

We believe our basic insight into unifying mutation changes with developer edits is also applicable to other key software testing realms. For example, mutation testing results for the old program version can optimize test selection [19] and prioritization [41] for the new version, because the potential impact of program edits can be simulated by existing mutants. We plan to establish these connections in future work.

This paper makes the following contributions:

- We **unify two widely used dimensions of software changes**: mechanical mutation changes and developer edits. This paper leverages this unified view to calculate the spectra as well as impacts of program edits to localize faults for evolving software. Furthermore, this unified view can also impact other realms of software testing.
- We **present the** FIFL **fault localization framework** to improve the accuracy of state-of-the-art techniques for localizing failure-inducing edits using the existing mutation testing results on the old program version. This framework creates a new dimension of possibilities to improve fault localization during software evolution.
- We **present an empirical study** on the code repositories of nine real-world Java programs. The experimental results show that FIFL (using its default settings) is able to outperform the state-of-the-art FAULTTRACER technique significantly (e.g., by more than 80% for some subjects) in localizing failure-inducing edits, indicating a promising future for localizing faulty edits by injecting mechanical faults.

2. Example

In this section, we use the example in Figure 1 to illustrate the FAULTTRACER approach for localizing failure-inducing edits and to motivate our FIFL approach. Figure 1 (a) shows the edited program, which manages the basic bank account functionality of two account types, i.e., `BankAcnt` (basic account type), `SuperAcnt` (super account type). Figure 1 (b) presents the regression test suite for validating the edits made on the example program. Assuming that the developer made a failure-inducing edit when adding `SuperAcnt.deposit()`[2] (shown in gray), `test4` then fails and detects the fault. The goal is to identify the failure-inducing edit precisely. We first show the steps applied by FAULTTRACER, then we show the limitations of FAULTTRACER and the intuition of FIFL.

[1] For evolving software systems, the mutation testing results for the old version may be already available in the repository, and ready to use.

[2] Please note that in this paper we omit the parameters and return types for methods to make the method names shorter.

```
1 public class BankAcnt{
2   public static String bank="ABank"
3   public String account;
4   public double saving=100;
5   public double iRate=0.01;
6   public double iRate2=0.02;
7   public Acnt(String a){account=a;}
8   public double getBalance()
9   {return saving;}
10  public double withdraw (double v) {
11    if(saving>=v) {
12      saving = saving-v;
13      return v;
14    }else return 0;
15  }
16  public void deposit (double v)
17  {saving = saving+v;}
18  double getRate(){return iRate;}
19  double getRate2(){return iRate2;}
20 }
21 public class SuperAcnt extends BankAcnt {
22   public double iRate=0.03;
23   public SuperAcnt(String a){super(a);}
24   public void deposit(double v) {
25     //fault, "0" should be "v"
26     saving=saving+0;
27     if(v>=50){saving=saving+1.0;}
28   }
29 }
30          (a)
```

```
1 public class TestSuite{
2   void test1() {
3     BankAcnt acnt=new BankAcnt("acnt1");
4     acnt.withdraw(20);
5     double rate=acnt.getRate();
6     assertEquals(acnt.getBalance(), 80);
7   }
8   void test2() {
9     SuperAcnt acnt=new SuperAcnt("acnt1");
10    acnt.withdraw(40);
11    assertEquals(acnt.getBalance(), 60);
12  }
13  void test3() {
14    SuperAcnt acnt=new SuperAcnt("acnt1");
15    acnt.deposit(0);
16    assertEquals(acnt.getBalance(), 100);
17  }
18  void test4() {
19    BankAcnt acnt1=new BankAcnt("acnt1");
20    SuperAcnt acnt2=new SuperAcnt("acnt2");
21    double amount=acnt1.withdraw(80);
22    double rate1=acnt1.getRate();
23    acnt2.deposit(amount);
24    double rate2=acnt2.getRate();
25    assertEquals(acnt2.getBalance(), 180);
26  }
27 }
28          (b)
```

Figure 1. (a) Example in evolution. Note that methods/fields in box are added, methods/fields with line-through are deleted, and methods/fields with underlines are changed. The statements with underlines inside changed methods are added. (b) Tests for the example.

Following traditional change impact analysis [40, 57], FAULTTRACER first extracts the edits between program versions as *atomic changes*, denoted as Δ. Atomic changes are categorized as added methods (AM), deleted methods (DM), changed methods (CM), added fields (AF), deleted fields (DF), changed instance fields (CFI), changed static fields (CSFI), field lookup changes (LC_f) due to the field hiding hierarchy changes, and method lookup (i.e., dynamic dispatch) changes (LC_m) due to the method overriding hierarchy changes. Note that FAULTTRACER splits all higher-level changes (e.g., class changes) into atomic changes. FAULTTRACER also infers dependences between atomic changes. For example, a method/field lookup change is dependent on the method/field addition or deletion that causes the lookup change. A non-lookup change c_1 (e.g., CM or AM) is dependent on another atomic change c_2 iff applying c_1 to the original program version without applying c_2 results in a syntactically invalid program. FAULTTRACER extracts atomic changes AM(SuperAcnt.deposit()), LC_m(SuperAcnt, SuperAcnt.deposit())[3], DF(BankAcnt.iRate2), etc, for the example in Figure 1. For the change dependences, DF(BankAcnt.iRate2) is determined to be dependent on DM(BankAcnt.getRate2()) (DM(BankAcnt.getRate2()) $\preceq$ DF(BankAcnt.iRate2)), as deleting BankAcnt.iRate2 without deleting method BankAcnt.getRate2() can cause BankAcnt.getRate2() to access a field without definition. FAULTTRACER also infers that LC_m(SuperAcnt, SuperAcnt.deposit()) depends on method addition AM(SuperAcnt.deposit()) (AM(SuperAcnt.deposit()) $\preceq$ LC_m(SuperAcnt, SuperAcnt.deposit())), since the AM change causes the LC_m change.

Second, FAULTTRACER determines the set of *affected tests* in the regression suites that have been influenced by the program edits based on the precise Extended Call Graph (ECG) analysis [57]. For each affected test, FAULTTRACER further identifies the set of atomic changes that might have changed the test's behavior, and denotes them as *affecting changes* for the test. For the example program, all the four tests are affected tests, and their affecting changes are shown in Columns 3-6 in the upper part of Table 1. A checkmark denotes that an atomic change is an affecting change of a affected test.

Third, FAULTTRACER uses the correlation between tests and affecting changes to determine the potential failure-inducing edits. The basic intuition is that an affecting change that is mainly executed by failed tests rather than passed tests is more suspicious. Therefore, FAULTTRACER utilizes the correlation between affecting changes and the failed tests to calculate the suspiciousness score for each affecting change.

[3] An LC_m change LC_m(R, S.m()) models the fact that an invocation to method S.m() on an object with run-time type R results in a different target method due to method additions or deletions during evolution.

Edits	Mapping Mutants	Affected Tests				Suspiciousness Score			
		test1	test2	test3	test4	Tarantula	SBI	Jaccard	Ochiai
`CFI(BankAcnt.iRate)`		✓			✓	0.75	0.50	0.50	0.71
`CM(BankAcnt.withdraw())`		✓	✓		✓	0.60	0.33	0.33	0.58
`AF(SuperAcnt.iRate)`					✓	1.00	1.00	1.00	1.00
`AM(SuperAcnt.deposit())`				✓	✓	0.75	0.50	0.50	0.71
`CFI(BankAcnt.iRate)`	line 5: `BankAcnt.iRate=1.0`					0.00	0.00	0.00	0.00
	line 5: `BankAcnt.iRate=0.0`					0.00	0.00	0.00	0.00
	line 5: `BankAcnt.iRate=-1.0`					0.00	0.00	0.00	0.00
`CM(BankAcnt.withdraw())`	line 12: `saving = saving+v;`	✓	✓			0.00	0.00	0.00	0.00
	line 12: `saving = saving/v;`	✓	✓			0.00	0.00	0.00	0.00
	line 12: `saving = saving*v;`	✓	✓			0.00	0.00	0.00	0.00
	line 12: `saving = saving%v;`	✓	✓			0.00	0.00	0.00	0.00
`AF(SuperAcnt.iRate)`	line 5: `BankAcnt.iRate=1.0`					0.00	0.00	0.00	0.00
	line 5: `BankAcnt.iRate=0.0`					0.00	0.00	0.00	0.00
	line 5: `BankAcnt.iRate=-1.0`					0.00	0.00	0.00	0.00
`AM(SuperAcnt.deposit())`	line 17: `saving = saving-v;`				✓	1.00	1.00	1.00	1.00
	line 17: `saving = saving/v;`			✓	✓	0.75	0.50	0.50	0.71
	line 17: `saving = saving*v;`			✓	✓	0.75	0.50	0.50	0.71
	line 17: `saving = saving%v;`			✓	✓	0.75	0.50	0.50	0.71
Out		P	P	P	F				

Table 1. Suspiciousness Calculation for Developer Edits and Mutation Changes.

Columns 7-10 of Table 1 show the suspiciousness score for each affecting change calculated by four well-known suspicious calculation formulae, i.e., Tarantula [23], Statistical Bug Isolation (SBI) [28], Jaccard [1], and Ochiai [1, 52]. However, the localization results are not ideal for this example: all the four formulas rank the real failure-inducing edit `AM(SuperAcnt.deposit())` as the tied third suspicious edit. The reason is that `AM(SuperAcnt.deposit())` is executed by `test3`, which passed, and `AF(SuperAcnt.iRate)` happens to be executed by the only failed test, thus mistakenly lowering the rank of `AM(SuperAcnt.deposit())` and lifting the rank of `AF(SuperAcnt.iRate)`.

The basic intuition of FIFL is that the mutation changes made by mutants of the old program version are able to simulate the impacts of developers' edits, and make the test execution results on some suspicious mutants (which share the same locations with real failure-inducing edits) similar to the test execution results on the new program version (with program edits). Therefore, we can directly use the existing mutation testing results of the old program version to boost the fault localization results while avoiding the drawbacks of iteratively applying subsets of program edits. For example, we can use the mutants occurring on the statements inside the same code element with `CFI(BankAcnt.iRate)` or `CM(BankAcnt.withdraw())` to simulate the effect of these two edits. For `AM(SuperAcnt.deposit())` and `AF(SuperAcnt.iRate)`, we cannot find the statements that share the same code elements with them because they do not exist on the old version. After analyzing the program, we find that a mutant occurring in `BankAcnt.deposit()` has a similar impact with adding `SuperAcnt.deposit()`, because the invocation to `SuperAcnt.deposit()` results in the target method of `BankAcnt.deposit()` in the old version. Therefore, adding `SuperAcnt.deposit()` may have a similar impact with changing `BankAcnt.deposit()` (using mutation). Similarly, we find that a mutant occurring in `BankAcnt.iRate` has a similar effect with editing a `SuperAcnt.iRate`, since `SuperAcnt.iRate` hides `BankAcnt.iRate` in the new version (detailed change mapping inference is shown in Section 3.1). For each edit, Column 2 of the lower part of Table 1 shows some example mapping mutants that may simulate each edit. Columns 3-6 show the test execution results for each mutant of the old program version. A checkmark denotes a mutant is killed by a test, i.e., the test fails on the mutant. Intuitively, we can find that any mapping mutants of the first three edits cannot fail the real failed test, `test4`, while a mapping mutant (in `BankAcnt.deposit()`) of the real failure-inducing edit, `AM(SuperAcnt.deposit())`, has exactly the same test execution results with the test execution results after evolution, indicating the benefits of improving edit suspiciousness calculation based on mutation testing. The detailed fault localization for this example will be further illustrated in Section 3.3.

3. Approach

Figure 2 shows the general framework of FIFL. Assume we have two program versions during software evolution, P and P', together with their regression test suite T, which passes on P but failed on P'. First, traditional mutation testing process is applied on P: generating all the mutants M for P and recording the mutant execution results, i.e., the correlation between mutants and the tests that kill them (denoted as MT). As FIFL only requires mutation testing results on the old version, FIFL recommends that this step is performed in the background in parallel with developing the new version, and thus the mutation testing results are directly available before applying FIFL. Second, FIFL extracts edits between P

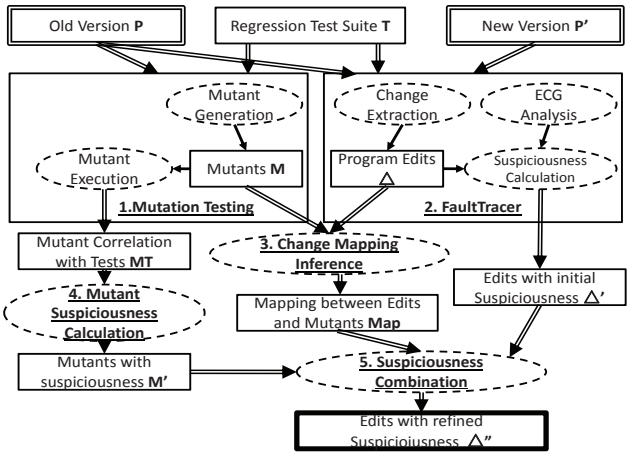


Figure 2. FIFL architecture.

and P' and calculates the suspiciousness of each edit using FAULTTRACER [57], which utilizes the spectrum information of edits and assigns a higher suspiciousness value to a edit if it is mainly executed by failed tests. Third, FIFL infers the mapping between developer edits and mutants based on a set of inference rules. Fourth, FIFL calculates the suspiciousness value for each mutant. A mutant is assigned with a higher suspiciousness value if it has a similar impact to regression tests with the program after edits, P'. Finally, FIFL refines the suspiciousness values of program edits (based on spectrum information) using the suspiciousness of their mapping mutants (based on impact information), and returns the edits with final suspiciousness values, Δ'', as the final result. As the first two steps (i.e., mutation testing and FAULTTRACER) are based on existing techniques, the following subsections show the change mapping inference (Section 3.1), mutant suspicious calculation (Section 3.2), and suspiciousness combination (Section 3.3) in detail.

3.1 Change Mapping Inference

In order to bridge the developer edits between P and P' with the mechanical changes to P via mutation, FIFL defines a set of inference rules for inferring the mapping between them. Figure 3 shows the inference rules. In the figure, m_c denotes the corresponding method for method-level change c, f_c denotes the corresponding field for field-level change c, and s_μ denotes the mutated statement of a mutant μ. $s \sqsubset_P x$ denotes that statement s is within the scope of method or field x in the old program version P. $f \dashrightarrow_{P'} f'$ denotes that field f hides field f' in the new version P', and $m \rightarrow_{P'} m'$ denotes that method m overrides method m' in P'. $c \preceq c'$ denotes change c' depends on change c. Finally, $c \mapsto \mu$ denotes that edit c is mapped with mutant μ. To better motivate and illustrate the rules, we present simple artificial examples as well as real-world code snippets to show how the change mapping can help increase the suspiciousness of

```
    public StringBuffer format(Calendar calendar,
        StringBuffer buf) {
866:    if (mTimeZoneForced) {
867:        calendar.getTimeInMillis();
868:        calendar = (Calendar) calendar.clone();
869:        calendar.setTimeZone(mTimeZone);
870:    }
871:    return applyRules(calendar, buf);
    }
```

Figure 4. Code snippet of *Commons-Lang* V3.03 to illustrate change mapping for CM edits

failure-inducing edits and/or decrease the suspiciousness of fault-free edits.

3.1.1 Inference for Changed/Deleted Elements

Shown by the first two rules, for modifications and deletions of methods and fields, the mapping is trivial: FIFL just maps a change with a mutant if the change and the mutant occur in the same method or field (since the method or field exists in the old version). The mapped developer edits and mutation changes occur at the same functional point and thus these two dimensions of changes may have similar impacts to the program under test.

For example, when *Commons-Lang* evolves from V3.02 to V3.03, the developer changed the `format()` method of class `FastDateFormat` and removed lines 866 to 870. As the changed method is executed by 35 tests with only one failed, making traditional approaches based on spectrum information not able to localize the fault precisely. In contrast, FIFL directly maps the `CM` change to the 5 mutants occurred inside the `format()` method in V3.02. Within the mapped mutants, 3 mutants, which remove method invocations for line 867 to line 869 respectively, have exactly the same failed test as V3.03, demonstrating that the mapped mutants can be used to simulate the effect of real method changes. This mapping significantly increase the ranking of the failure-inducing edit (details shown in Section 5).

3.1.2 Inference for Added Elements Overridding/Hiding Existing Elements

The mutant mapping inference rules for additions of fields and methods are more complex because they have no corresponding code elements in the old version. We illustrate those rules with examples in Figure 6. In the figure, we connect an added element (`AM` or `AF`) with another non-added element using a twin line if and only if the added element can be mapped to the mutant within the scope of the non-added element. The basic intuition for Rules 3-6 is that the mutants occurring in a method/field c' that is close to the added method/field c in the overriding/hiding hierarchy (such that an invocation/access to c actually executes c' in the old version) can be used to simulate the impact of adding c, because both adding c and mutating c' may change the execution of c'. Rules 3 and 4 define the mapping inference for method additions that override some methods: If the added method

$$\frac{c \in CM \cup DM \;\; \mu \in M \;\; s_\mu \sqsubset_P m_c}{c \mapsto \mu}(1)$$

$$\frac{c \in CFI \cup CSFI \cup DF \;\; \mu \in M \;\; s_\mu \sqsubset_P f_c}{c \mapsto \mu}(2)$$

$$\frac{c \in AM \;\; \mu \in M \;\; s_\mu \sqsubset_P m \;\; m_c \rightarrow_{P'} m}{c \mapsto \mu}(3) \qquad \frac{c, c' \in AM \;\; \mu \in M \;\; c' \mapsto \mu \;\; m_c \rightarrow_{P'} m_{c'}}{c \mapsto \mu}(4)$$

$$\frac{c \in AF \;\; \mu \in M \;\; s_\mu \sqsubset_P f \;\; f_c \dashrightarrow_{P'} f}{c \mapsto \mu}(5) \qquad \frac{c, c' \in AF \;\; \mu \in M \;\; c' \mapsto \mu \;\; f_c \dashrightarrow_{P'} f_{c'}}{c \mapsto \mu}(6)$$

$$\frac{c \in AM \;\; c' \in DM \;\; c'' \in LC_m \;\; \mu \in M \;\; s_\mu \sqsubset_P m_{c'} \;\; c \preceq c'' \;\; c' \preceq c''}{c \mapsto \mu}(7)$$

$$\frac{c \in AF \;\; c' \in DF \;\; c'' \in LC_f \;\; \mu \in M \;\; s_\mu \sqsubset_P f_{c'} \;\; c \preceq c'' \;\; c' \preceq c''}{c \mapsto \mu}(8)$$

$$\frac{c \in AM \cup AF \;\; c' \in \Delta \;\; \mu \in M \;\; c' \mapsto \mu \;\; c \preceq c'}{c \mapsto \mu}(9)$$

Figure 3. Rules for inferring change mapping.

```
class org.joda.time.chrono.AssembledChronology:
public long getDateTimeMillis(int year,
    int monthOfYear, int dayOfMonth, int hourOfDay,
    int minuteOfHour, int secondOfMinute, int
        millisOfSecond) {...}
►class org.joda.time.chrono.BasicChronology:
  public long getDateTimeMillis(int year,
      int monthOfYear, int dayOfMonth, int hourOfDay,
      int minuteOfHour, int secondOfMinute, int
          millisOfSecond) {...}
```

Figure 5. Code snippet of *Joda-Time* V1.20 to illustrate change mapping for AM edits with overridden methods

overrides some existing methods that are not newly added, Rule 3 maps the addition change to all mutants that occur inside the existing method; if the atomic change of the method overridden by the added method is already mapped with a set of mutants, Rule 4 also maps the added method with those mutants. Similarly, Rules 5 and 6 infer the change mapping for field additions that hide other fields: If the added field hides some existing fields that are not newly added, Rule 5 maps the addition change to all mutants that occur inside the existing field; otherwise, if the change on the field hidden by the added field is already mapped with a set of mutants, Rule 6 also maps the added field to those mutants. Note that Rules 4 and 6 should be iteratively applied until they reach a fix point. To illustrate, the change mapping inference steps for Figure 6(a) are shown as follows:

Applied Rules	AM(m_3)	AM(m_4)
Rule 3	M_{m_2}	-
Rule 4	M_{m_2}	M_{m_2}

where M_{m_2} denotes the mutants occurring in the body of method m_2. Similarly, the two AF changes in Figure 6(b) are mapped with mutants inside f_1 and f_3.

For example, when *Joda-Time* evolves from V1.10 to V1.20, the fault-free edit AM(getDateTimeMillis()) in class BasicChronology was ranked high because it was executed by some failed tests accidentally. As the method was newly added, Rule 1 cannot map the edit with any mutants of the old version. Figure 5 shows the class inheritance hierarchy for the class containing the added method. In the figure, ► denotes the below class is a subclass of the above class. As the added method overrides an existing method in class AssembledChronology (denoting that the two methods have similar functionalities), the invocation to the specific functionality of the new method may be resolved to the overridden method in the old version. Thus, some mutation changes to the old overridden method may have similar impacts with the edit of adding the faulty method, and thus the mutants in the overridden method can be mapped to the AM edit to simulate its impact. In fact, using this mapping, FIFL successfully decreases the suspiciousness of the fault-free edit.

3.1.3 Inference for Added Elements Sharing Overriding/Hiding Hierarchy with Deleted Elements

There may also be some added method/field c that shares the same overriding/hiding hierarchy with some deleted method-/field c', i.e., although they never co-exist in one version, they may implement the same functionality. Therefore, the mutants within $m_{c'}$ in the old version may also be able to simulate the impact of c. As added and deleted elements do not exist in the same version, FIFL cannot use the ordinary overriding/hiding hierarchy analysis to infer the change mapping. Instead, FIFL utilizes the fact that both addition changes and deletion changes would cause method or field lookup changes, and uses those lookup changes to bridge the mapping between addition changes and mutants in deleted elements. For any method/field addition c, if some method/-field deletion c' causes the same method/field lookup change with c, Rule 7/8 maps the mutants inside the corresponding deleted element of c' with c. Figures 6(c) and 6(d) illustrate that the mutants within the deleted methods/fields can be mapped with addition changes.

In the same revision with the last example code snippet (i.e., when *Joda-Time* evolves from V1.10 to V1.20), the fault-free program edit AM(getAverageMillisPerMonth()) in class BasicFixedMonthChronology was ranked high by the existing FAULTTRACER approach, because it was executed by the failed tests accidentally. As the added method does not override any existing method, Rules 3 and 4 can-

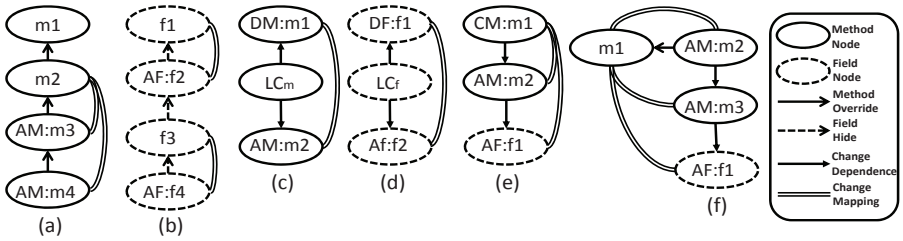


Figure 6. Illustration for mutant mapping.

```
class org.joda.time.chrono.AssembledChronology
▶class org.joda.time.chrono.BaseGJChronology
  ▶class org.joda.time.chrono.CopticChronology:
    long getAverageMillisPerMonth() {...}
```

(a) *Joda-Time* V1.10

```
class org.joda.time.chrono.AssembledChronology
▶class org.joda.time.chrono.BasicChronology
  ▶class org.joda.time.chrono.BasicFixedMonthChronology:
    long getAverageMillisPerMonth() {...}
    ▶class org.joda.time.chrono.CopticChronology
```

(b) *Joda-Time* V1.20

Figure 7. Code snippets of *Joda-Time* V1.10 and V1.20 to illustrate change mapping of AM edits with deleted methods sharing the same overriding hierarchy

not map the edit with any mutant. Figure 7(b) shows the inheritance hierarchy for the class of the added method (`BasicFixedMonthChronology`). The containing class of the edit has a superclass named `AssembledChronology` and a subclass named `CopticChronology`. Shown in Figure 7(a), the old version has a method (which was deleted in the new version) with the same signature and under the same class inheritance hierarchy as the added method. The actual change logic is that the developer pulled up the deleted method to a new superclass in the new version. In this way, the old deleted method and the new added method implement the same functionality and thus the mutation changes to the deleted method and the edit for adding the new method may have the same impact to the program. Therefore, using Rule 7, FIFL identifies that both the deleted method and the newly added method cause the same LC change: $\texttt{LC}_m$`(CopticChronology, *.getAverageMillisPerMonth())` (i.e., invocation of method `getAverageMillisPerMonth()` on run-time object of type `CopticChronology` resolves to a different method), and thus maps the mutants occurring on the old method with the method addition. In fact, the mutants for the old method have different test execution results with the new version, making the ranking of the fault-free `AM(getAverageMillisPerMonth())` decreased, and thus the ranking of actual failure-inducing edits increased.

```
    public XStream(ReflectionProvider...) {
        ...
367:    this.mapper=mapper==null?buildMapper():mapper;
        ...
    }
    private Mapper buildMapper() {
379:    Mapper mapper = new DefaultMapper(
     classLoaderReference);
380:    if ( useXStream11XmlFriendlyMapper() ){
381:        mapper = new XStream11XmlFMapper(mapper);
382:    }
        ...
    }
```

Figure 8. Code snippet of *XStream* V1.21 to illustrate change mapping for AM edits without methods sharing the same method overriding hierarchy

3.1.4 Inference for Other Added Elements

For the other added method/field c that neither overrides/hides any existing method/field nor shares the overriding/hiding hierarchy with any deleted method/field, if c is executed by the regression test suite, c must be invoked/accessed by some changed or added method c'. Then, a mutant μ occurring in c' may be used to simulate the impacts of c, because mutant μ in c' may have the same impacts with adding invocation to c in c'. In this situation, the change impact analysis component of FIFL would have detected that change c' depends on change c (i.e., $c \preceq c'$), because c' would invoke/access undeclared method/field without applying change c. Rule 9 then directly maps any mutant that has been mapped with c' to c. Note that Rule 9 should be iteratively applied until it arrives a fix point. To illustrate, the change mapping inference for Figure 6(e) is shown as follows:

Applied Rules	AM(m_2)	AF(f_1)
Rule 9	$Map[\texttt{CM}(m_1)]$	-
Rule 9	$Map[\texttt{CM}(m_1)]$	$Map[\texttt{CM}(m_1)]$

where $Map[\texttt{CM}(m_1)]$ denotes the mutants that have been mapped to change $\texttt{CM}(m_1)$ (Rule 1). The first inference using Rule 9 does not find mapped mutants for $\texttt{AF}(f_1)$, because $\texttt{AM}(m_2)$ has no mapped mutants yet. On the contrary, the second application of Rule 9 successfully finds mapped mutants for $\texttt{AM}(f_1)$, because $\texttt{AM}(m_2)$ is mapped to mutants at the first step.

Among the studied subjects, when *XStream* evolves from V1.20 to V1.21, the developer added the faulty method

`buildMapper()`. Shown in Figure 8, the added faulty method `buildMapper()` was invoked by a changed method `XStream()` at line 367 (which is the only changed line for the method, and invoked an old method for building mappers in the old version). The faulty `AM` change is executed by every tests because it is used for initializing mappers, and thus cannot be distinguished from other edits. Because the functionality of building mappers in the old version was also invoked by the changed method, some mutation changes (especially mutation changes on the specific line for invoking the mapper builder) may have similar impacts as adding a new method for building mappers. Thus, FIFL maps the `AM` edit with the mutants that occurred in the old version of `XStream()` based on Rules 1 and 9. In fact, some mutants that occurred at the changed line of the old method `XStream()` exactly fail the same tests with the new version because the statement for constructing the old map builder was changed, demonstrating that mapped mutants can be used to simulate the effect of method additions when the added methods do not override any existing method or share overriding hierarchy with any deleted method.

Finally, Figure 6(f) illustrates a slightly more complex situation: an added method m_2 overrides an existing method m_1, and also invokes another added method m_3, which in turns accesses an added field f_1. FIFL first maps `AM`(m_2) with mutants in m_1 based on Rule 3, then maps `AF`(f_1) and `AM`(m_3) with mutants in m_1 based on Rule 9. The detailed inference is shown as follows:

Applied Rules	`AM`(m_2)	`AM`(m_3)	`AF`(f_1)
Rule 3	M_{m_1}	-	-
Rule 9	M_{m_1}	M_{m_1}	-
Rule 9	M_{m_1}	M_{m_1}	M_{m_1}

Note that each program edits will be applied with any applicable rules and may be mapped to mutants from various methods/fields. FIFL uses all those mutants to select the most suitable one (details are shown in the next section).

3.2 Mutant Suspiciousness Calculation

The calculation of mutant suspiciousness is based on the intuition that a mutant that has a similar test pass/fail results with the program after edits might share same positions with the real failure-inducing program edits. Therefore, the suspiciousness of a mutant can be calculated based on the similarity between its test execution results and the test execution results after edits.

As mutant suspiciousness will be used to refine edit suspiciousness, we use the same suspicious calculation formulae used by FAULTTRACER, for calculating the suspiciousness of edits [57]. The difference is that FIFL additionally uses the correlation between tests and mutant killing as input, while FAULTTRACER only uses the correlation between tests and edits as input. FIFL adapts four representative suspiciousness computations for mutants as follows.

(1) *Statistical Bug Isolation.* Liblit et al. [28] first proposed Statistical Bug Isolation (SBI) to rank faulty predicates. Yu et al. [52] adapted SBI to rank potential faulty statements. FAULTTRACER adapted the SBI formula to calculate the suspiciousness for program edits [57]. We further adapt the formula to define a suspiciousness score for a mutant μ as $S_s(\mu)$:

$$\underbrace{(|K(\mu,T')\cap T'_f|)}_{failed(\mu)}/\underbrace{(|K(\mu,T')\cap T'_p|}_{passed(\mu)}+\underbrace{|K(\mu,T')\cap T'_f|)}_{failed(\mu)}$$

In this formula, T' denotes all affected tests, T'_p denotes the passed affected tests, T'_f denotes the failed affected tests, $K(\mu,T')$ denotes the affected tests that kill μ, $failed(\mu)$ denotes the number of failed affected tests that kill mutant μ, and $passed(\mu)$ denotes the number of passed affected tests that kill μ.

(2) *Tarantula.* Jones et al. [23] proposed Tarantula, which assigns higher suspiciousness scores to statements primarily executed by failed tests than statements primarily executed by passed tests. FAULTTRACER then adapted the Tarantula formula to calculate the suspiciousness for program edits [57]. Similarly, we adapt the formula to define a suspiciousness score for a mutant μ as $S_t(\mu)$:

$$\underbrace{(|K(\mu,T')\cap T'_f|/|T'_f|)}_{\%failed(\mu)}/\underbrace{(|K(\mu,T')\cap T'_p|/|T'_p|}_{\%passed(\mu)}+\underbrace{|K(\mu,T')\cap T'_f|/|T'_f|)}_{\%failed(\mu)}$$

In this formula, $\%failed(\mu)$ denotes the ratio of failed affected tests that can also kill mutant μ to all failed affected tests, while $\%passed(\mu)$ denotes the ratio of passed affected tests that can kill mutant μ to all passed affected tests.

(3) *Ochiai.* Yu et al. [52] and Abreu et al. [1] used the Ochiai formula, which originated from the molecular biology domain, to rank faulty statements in one specific program version. FAULTTRACER then adapted the Ochiai formula to calculate the suspiciousness for program edits [57]. Similarly, we adapt the formula to define a suspiciousness score for a mutant μ as $S_o(\mu)$:

$$\underbrace{(|K(\mu,T')\cap T'_f|)}_{failed(\mu)}/\sqrt{\underbrace{|T'_f|}_{all\_failed}*\underbrace{(|K(\mu,T')\cap T'_p|}_{passed(\mu)}+\underbrace{|K(\mu,T')\cap T'_f|)}_{failed(\mu)}}$$

In this formula, all_failed denotes the number of all failed affected tests.

(4) *Jaccard.* Abreu et al. [1] used the Jaccard formula, which was used for measuring the statistical similarity and diversity between sample sets, to rank faulty statements. FAULTTRACER then adapted the Jaccard formula to calculate the suspiciousness for program edits [57]. We further adapt the formula to define a suspiciousness score for a mutant μ as $S_j(\mu)$:

$$\underbrace{(|K(\mu,T')\cap T'_f|)}_{failed(\mu)}/(\underbrace{|T'_f|}_{all\_failed}+\underbrace{|K(\mu,T')\cap T'_p|)}_{passed(\mu)}$$

In this way, the suspiciousness values for all mutants are between 0.00 and 1.00. If a mutant failed exactly the same set of tests with the program after edits, its suspiciousness would be calculated as 1.00 by all formulae. To illustrate, we show the suspiciousness score calculated for each mutant of the example program in Columns 7-10 of the lower part of Table 1. Shown in the gray row, FIFL directly determines a mapping mutant of the real failure-inducing edits as the most suspicious. Note that when a real program has multiple faulty edits, the suspiciousness of mapped mutants for all the faulty edits can be determined as suspicious because the injection of each mutant may make the program fail a subset of the real failed tests. Therefore, all the four formulae can calculate those mutants mapped with faulty edits as suspicious.

3.3 Suspiciousness Combination

In this section, we present suspiciousness combination based on the suspiciousness for mutants, the suspiciousness for edits, and the mapping between edits and mutants. FIFL integrates three general combination strategies shown as follows. Note that FIFL uses the maximum suspiciousness value for the set of mapping mutants (i.e., using the most suitable mutant) to refine the suspiciousness of an edit, because a large ratio of mutants might not be effective in simulating the impacts of program edits.

(1) *Min-Max*. This strategy refines an edits's suspiciousness by using the minimum value between the edit's initial suspiciousness value and the maximum suspiciousness value for its mapping mutants:

$$S_{refined}(c) = Min(S(c), Max_{\mu \in Map[c]} S(\mu))$$

To illustrate, when we use Ochiai for both the edit and mutant suspiciousness calculation, the refined suspiciousness value for `AM(SuperAcnt.deposit())` is calculated as follows.

$$\begin{aligned} S_{refined} \quad & (\texttt{AM(SuperAcnt.deposit())}) = \\ & Min(0.71, Max(1.00, 0.71, 0.71, 0.71)) = 0.71 \end{aligned}$$

The suspiciousness value for other three edits are all refined to 0.00, making `AM(SuperAcnt.deposit())` as the top one suspicious edit.

(2) *Max-Max*. This strategy refines an edits's suspiciousness by FAULTTRACER using the maximum value between the edit's initial suspiciousness value and the maximum suspiciousness value for its mapping mutants:

$$S_{refined}(c) = Max(S(c), Max_{\mu \in Map[c]} S(\mu))$$

in which $Map[c]$ denotes all the mapping mutants for edit c. To illustrate, when we use Ochiai for both the edit and mutant suspiciousness calculation, the refined suspiciousness value for `AM(SuperAcnt.deposit())` is calculated as follows.

$$\begin{aligned} S_{refined} \quad & (\texttt{AM(SuperAcnt.deposit())}) = \\ & Max(0.71, Max(1.00, 0.71, 0.71, 0.71)) = 1.00 \end{aligned}$$

The suspiciousness value for other three edits are refined as 0.71, 0.58, and 1.00, making `AM(SuperAcnt.deposit())` tied as the top ranking edit.

(3) *Ratio-Max*. This strategy refines an edit's suspiciousness by assigning different weights to the edit's initial suspiciousness value and the maximum suspiciousness value of its mapping mutants. The combination is shown as follows.

$$S_{refined}(c) = \alpha * S(c) + (1 - \alpha) * Max_{\mu \in Map[c]} S(\mu)$$

where $\alpha \in [0.0, 1.0)$ denotes the weight for an edit's initial suspicious[4]. Note that when $\alpha = 0.0$, the strategy ranks edits only based on the suspiciousness of their mapping mutants. To illustrate, when we use Ochiai for both the edit and mutant suspiciousness calculation and the default α value of 0.5, the refined suspiciousness value for `AM(SuperAcnt.deposit())` is calculated as follows.

$$\begin{aligned} S_{refined} \quad & (\texttt{AM(SuperAcnt.deposit())}) = \\ & \alpha * 0.71 + \alpha * Max(1.00, 0.71, 0.71, 0.71) \\ & 0.5 * 0.71 + 0.5 * 1.00 = 0.86 \end{aligned}$$

The suspiciousness value for other three edits are refined as 0.36, 0.29, and 0.50, making `AM(SuperAcnt.deposit())` as the top one suspicious edit.

Note that when the number of mapping mutants for an edit is smaller than a threshold (2 in this paper), those mutants may not be sufficient to simulate the impact of the edit. Therefore, for this circumstance, FIFL simply keeps the initial suspiciousness for that edit.

3.4 Tackling the Cost of FIFL: Edit-Oriented Mutation Testing

Compared with the existing FAULTTRACER technique, FIFL additionally utilizes the available mutation testing results of the old program version from the repository to further refine the fault localization results. Mutation testing results can already be available due to its other applications, e.g., generating [8, 12, 18, 35, 56] or evaluating test suites [4, 42, 43]. In addition, since FIFL depends on the mutation testing results of the old version, the mutation testing process can be conducted at the same time with developing the new version, thus improving the fault localization results with no overhead.

However, the mutant testing results for the old program version might still be absent when doing fault localization, we further show how to tackle the cost of mutation testing at that situation. We propose the concept of *Edit-Oriented Mutation Testing*, which only collects mutation testing results of the mutants mapping with program edits, since the execution results of other mutants are not used by FIFL. Formally, the subset of mutants executed by edit-oriented mutation testing can be represented as follows.

$$M_{edit} = \{\mu | \forall c \in \Delta, \mu \in Map[c]\}$$

[4] We exclude the α value of 1.0, because the edit's suspiciousness is not refined at all in such a case.

Projects	Description	License	LoC(Src/Test)
Time&Money	Time and money library	MIT	2.7K/3.0K
Barbecue	Bar-code creator	BSD	5.4K/3.3K
Mime4J	Message stream parser	Apache2.0	7.0K/3.8K
Jaxen	Java XPath library	Apache-style	14.0K/8.8K
Xml-Security	XML security standards	MIT	19.8K/4.0K
XStream	Object serialization library	BSD	18.4K/20.1K
JMeter	Performance testing	Apache2.0	44.6K*
Commons-Lang	Java helper utilities	Apache2.0	23.3K/32.5K
Joda-Time	Time library	Apache2.0	32.9K/55.9K

∗ indicates source and test code are written together, and cannot be measured separately

Table 2. Subjects overview.

where Δ denotes all the edits between two program versions, and $Map[c]$ denotes the mapping mutants for program edit c. As the mutant generation is much more efficient than mutant execution [42], our implementation simply generates all the mutants, and only executes the mutants mapped with edits.

4. Implementation

We built our FIFL technique on top of Javalanche[5] [42] and FAULTTRACER [57]. FIFL uses Javalanche for the first step mutant generation and execution. Javalanche is a state-of-the-art mutation testing tool for Java programs. Javalanche allows efficient mutant generation, as well as mutant execution. More precisely, Javalanche uses a small set of sufficient mutant operators, and manipulates Java bytecode directly using *mutant schemata* to enable efficient mutant generation. In addition, Javalanche only executes the set of influenced tests for each mutant based on coverage checking, and allows parallel execution to enable efficient mutant execution.

FIFL uses FAULTTRACER for the second step edit detection and suspiciousness calculation. FAULTTRACER is a state-of-the-art technique for localizing failure-inducing program edits during software evolution. FAULTTRACER calculates the suspiciousness of each program edit based on their correlation with failed tests. FAULTTRACER has been shown to outperform the previous ranking technique [39] by more than 50% [57].

FIFL's third step change mapping inference requires the method-overriding hierarchy, field-hiding hierarchy, as well as source code scope for given changed entities, etc. We implemented this step based on the *Eclipse JDT toolkit*[6]. The fourth and the fifth steps mainly involve data computation and transformation, and are directly implemented with the Java language. Although FIFL is currently implemented for Java programs, the FIFL methodology of localizing faulty edits based on fault injection is generalizable for other object-oriented languages.

5. Experimental Study

FIFL aims to make suspiciousness calculation more precise for program edits. To evaluate the effectiveness of FIFL, the experimental study compares FIFL against FAULTTRACER [57], a state-of-the-art approach for localizing failure-inducing edits on real-world code repositories.

5.1 Independent Variables

According to the theory of experimentation, we used the following *independent variables* (IVs) to investigate their influences on the final experimental results:

IV1: Different Localization Approaches. We considered the following choices of approaches as the first independent variable: (1) FAULTTRACER, which is a state-of-the-art approach for localizing failure-inducing program edits; (2) FIFL, which is proposed in this paper and embodies the idea of injecting faults to localize failure-inducing edits.

IV2: Different Calculation Formulae for Edit Suspiciousness. We considered all the four formulae used by FAULTTRACER [57] to calculate the suspiciousness of program edits: (1) *SBI*; (2) *Tarantula*; (3) *Ochiai*; and (4) *Jaccard*.

IV3: Different Calculation Formulae for Mutant Suspiciousness. Similarly, we considered the same set of formulae for calculating the suspiciousness of mutants (shown in Section 3.2): (1) *SBI*; (2) *Tarantula*; (3) *Ochiai*; and (4) *Jaccard*.

IV4: Different Combination Strategies. We considered all the three combination strategies shown in Section 3.3 for refining the suspiciousness of edits based on the suspiciousness of mutants: (1)*Min-Max*; (2)*Max-Max*; and (3)*Ratio-Max*. For the *Ratio-Max* strategy, we use values of α ranging from 0.00 to 0.95 with increments of 0.05, i.e., 20 values of α. Note that when α=0.0, the strategy ranks edits based on pure mutant suspiciousness.

5.2 Dependent Variables

Since we are concerned with the effectiveness as well as efficiency achieved by our FIFL approach, we used the following dependent variable (DV):

DV: Rank of Failure-Inducing Edits. This variable denotes the total number of edits that developers need to inspect before finding the real failure-inducing edits when using the compared techniques.

5.3 Subjects and Experimental Setup

We obtained versions of the source code of nine open-source projects in various application domains, which have been widely used for regression testing and mutation testing research [10, 42, 43, 58]. Table 2 depicts brief information about the latest release of each studied project. The sizes of the studied projects range from 5,675 lines of code (LoC) (*Time&Money*, including 2,678 LoC source code and 2,997 LoC test code) to 88,835 LoC (*Joda-Time*, with 32,932 LoC source code and 55,903 LoC test code). We obtained *Xml-Security* and *JMeter* from the well-known Software-artifact Infrastructure Repository (SIR) [10], and all the other projects from their host repositories. For each project, we obtained all the available releases in its repository, and

[5] http://www.st.cs.uni-saarland.de/mutation/. Accessed in July 2013.

[6] http://www.eclipse.org/jdt/. Accessed in July 2013.

No	Project	Version Pair	#Tests	#FTests	#Edits	#FEdits	All Mutants		Mapped Mutants	
							Number	Execution Time	Number	Execution Time
P_1	*Time&Money*	3.0, 4.0	143	1	215	1	1737	9min24s	792	7min38s
P_2	*Time&Money*	4.0, 5.0	159	1	246	1	1984	6min43s	325	1min5s
P_3	*Barbecue*	1.5a1, 1.5a2	160	2	23	1	41310	15min37s	96	57s
P_4	*Mime4J*	0.50, 0.60	120	8	2862	3	19111	181min20s	8086	37min50s
P_5	*Mime4J*	0.61, 0.70	348	3	3160	4	27654	227min14s	24443	49min15s
P_6	*Jaxen*	1.0b7, 1.0b9	24	2	204	3	3820	100min25s	964	28min38s
P_7	*Jaxen*	1.1b2, 1.1b5	69	2	419	1	5489	19min8s	1493	7min42s
P_8	*Jaxen*	1.1b6, 1.1b7	243	2	473	5	9704	44min49s	6037	24min8s
P_9	*Jaxen*	1.1b9, 1.1b11	645	1	92	1	10045	61min59s	157	2min9s
P_{10}	*Xml-Security*	1.0, 2.0	91	5	329	2	10599	16min6s	1134	2min3s
P_{11}	*XStream*	1.20, 1.21	637	3	209	1	10956	49min51s	547	5min31s
P_{12}	*XStream*	1.21, 1.22	698	1	222	2	11516	54min24s	847	6min56s
P_{13}	*XStream*	1.22, 1.30	768	24	540	11	12536	64min22s	1870	8min25s
P_{14}	*XStream*	1.30, 1.31	885	12	416	3	14140	96min43s	2206	13min0s
P_{15}	*XStream*	1.31, 1.40	924	13	1225	7	15006	99min26s	3462	22min15s
P_{16}	*XStream*	1.41, 1.42	1200	6	136	5	18046	132min2s	1817	10min50s
P_{17}	*JMeter*	0.0, 1.0	51	1	1714	1	5363	29min56s	1779	8min40s
P_{18}	*JMeter*	1.0, 2.0	60	2	1056	1	21896	57min32s	3604	13min22s
P_{19}	*JMeter*	2.0, 3.0	72	11	2809	4	8067	43min27s	4347	34min44s
P_{20}	*JMeter*	3.0, 4.0	76	1	764	1	7116	34min36s	703	10min50s
P_{21}	*Commons-Lang*	3.02, 3.03	1698	1	221	1	20792	90min26s	803	4min7s
P_{22}	*Commons-Lang*	3.03, 3.04	1703	2	172	2	20792	90min29s	1441	3min33s
P_{23}	*Joda-Time*	0.90, 0.95	219	4	5976	2	6581	6min29s	5365	3min37s
P_{24}	*Joda-Time*	0.98, 0.99	1932	6	1254	2	16208	31min36s	3631	4min45s
P_{25}	*Joda-Time*	1.10, 1.20	2420	1	793	1	19012	53min10s	1997	12min32s
P_{26}	*Joda-Time*	1.20, 1.30	2516	11	571	3	19566	106min10s	718	31min32s

Table 3. Version pairs with test failures.

treated every two continuous releases as a version pair. For each version pair, we applied the regression test suite of the old version on the new version, and treated the edits that cause the regression suite to fail on the new version as regression faults. We studied all those version pairs with regression test failures to evaluate FIFL's performance. The experimental study was performed on a Dell desktop with Intel i7 8-Core Processor (2.8G Hz), 8G RAM, and Win7 Enterprise 64-bit version.

For all the projects, we were able to find version pairs with regression faults except *JMeter*. However, *JMeter* comes with seeded faults in SIR, thus we use seeded faults for *JMeter*. In total, we have 26 version pairs with regression faults, and the details are shown in Table 3. In Table 3, Column 1 shows the abbreviations for all the version pairs with regression faults. Columns 2 and 3 show the project name and corresponding versions for each version pair. Columns 4 and 5 show the number of tests and failed tests for each version pair. Columns 6 and 7 show the number of edits and failure-inducing edits for each version pair.

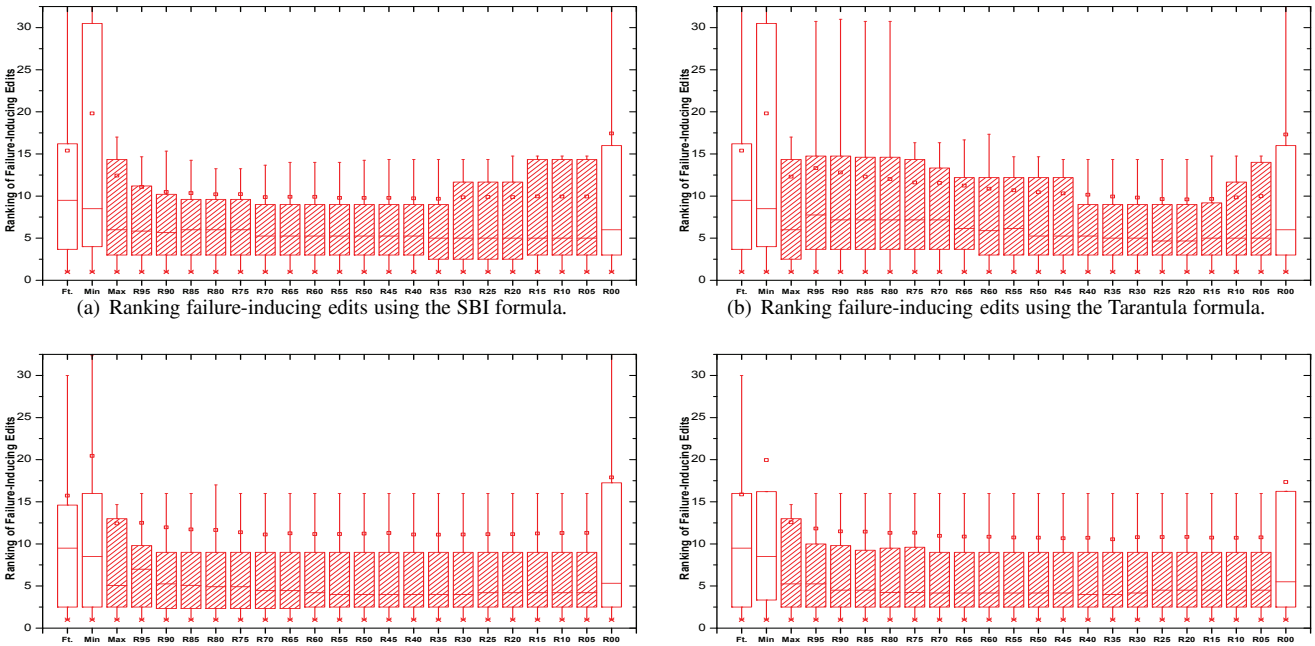
The table also presents the mutation testing statistics using Javalanche. Columns 8 and 9 show the number and execution time for all mutants. Similarly, Columns 10 and 11 show the number and execution time for the mutants that are actually needed by FIFL (i.e., mutants mapped with edits). We observe that overall mutation testing time by Javalanche for each studied version pair is acceptable for the studied subjects, ranging from 6 minutes 43 seconds to 227 minutes 14 seconds. The reason is that Javalanche embodies a set of optimization strategies for improving efficiency (Section 4). We also find that the mutation testing time for only mapped mutants is much more efficient, and is less than 1 hour for all subjects. Recall that FIFL never costs the developer the entire mutation testing time: (1) mutation testing results can already be in the repository before applying FIFL due to its other applications; (2) the mutation testing results can be collected at the same time as developing the new version, because FIFL uses the mutation testing results on the old version; (3) even when mutation testing results are not available before applying FIFL, developers can only collect the mutation testing results for the mutants mapped with edits (Section 3.4).

5.4 Results and Analysis

In this section, we first compare all strategies of FIFL with FAULTTRACER (Section 5.4). Then we compare the default strategies of FIFL with FAULTTRACER in detail (Section 5.4.2). Finally, we discuss about the scope and limitations of the FIFL approach and its evaluation (Section 5.4.3).

5.4.1 Overall comparison between FAULTTRACER and various strategies of FIFL

Figures 9(a) to 9(d) show the comparison of FIFL with FAULTTRACER for different suspicious calculation formulae. We denote FAULTTRACER (the ranking based on pure edit suspiciousness) as *Ft.*, FIFL's *Min-Max* strategy as *Min*, and FIFL's *Max-Max* strategy as *Max*. For FIFL's *Ratio-Max* strategies, we use *R* and the value of α to represent each strategy. For example, we use *R95* to denote the *Ratio-Max* strategy with α=0.95. In each figure, the horizontal axis shows compared techniques, and the vertical axis shows the rank of failure-inducing edits by each technique across all the version pairs with regression faults. Each box plot shows



(a) Ranking failure-inducing edits using the SBI formula.
(b) Ranking failure-inducing edits using the Tarantula formula.
(c) Ranking failure-inducing edits using the Ochiai formula.
(d) Ranking failure-inducing edits using the Jaccard formula.

Figure 9. Ranking failure-inducing edits using various techniques with various formulae.

the average (a dot in the box), median (a line in the box), and upper/lower quartile values for the ranking of failure-inducing edits across various version pairs[7]. We mark all the techniques that outperform the original FAULTTRACER technique (which is based on the pure edit suspiciousness) in terms of both average and median values as shadowed box plots. The key findings from the experimental results are as follows.

First, in terms of median effectiveness across all version pairs, all ranking techniques of FIFL outperform FAULTTRACER. When using the SBI formula (the median case for the Tarantula formula is similar), in the median case, FAULTTRACER localizes failure-inducing edits within 9.5 edits. In contrast, the *Min-Max* and *Max-Max* strategies of FIFL localize failure-inducing edits within 8.5 and 6 edits, respectively. The *Ratio-Max* strategies of FIFL with all α values within $[0.00, 0.95]$ localize failure-inducing edits within 6 edits. When using the Ochiai formula (the median case for the Jaccard formula is similar), in the median case, FAULTTRACER localizes failure-inducing edits within 9.5 edits. In contrast, the *Min-Max* and *Max-Max* strategies of FIFL localize failure-inducing edits within 8.5 and 5.05 edits, respectively. Furthermore, the *Ratio-Max* strategies of FIFL with $\alpha \in [0.30, 0.55]$ localize failure-inducing edits within 4 edits, an improvement of 57.89% over FAULTTRACER.

Second, in terms of average effectiveness across all version pairs, the *Max-Max* strategy and the *Ratio-Max* strategies with $\alpha \in [0.05, 0.95]$ still outperform FAULTTRACER. For example, using the SBI formula, FAULTTRACER localizes failure-inducing edits within 15.40 edits, the *Max-Max* strategy of FIFL localizes failure-inducing edits within 12.42 edits, and all strategies of FIFL with $\alpha \in [0.05, 0.95]$ are able to localize faulty edits within 11.08 edits. Furthermore, the *Ratio-Max* strategy of FIFL with $\alpha = 0.35$ localizes failure-inducing edits within 9.68 edits, indicating an average improvement of 37.14% over FAULTTRACER. However, the *Min-Max* strategy and the *Ratio-Max* strategy with $\alpha = 0.00$ (the ranking based on pure mutant suspiciousness) cannot outperform FAULTTRACER. For example, when using the SBI formula, the *Min-Max* strategy and the *Ratio-Max* strategy with $\alpha = 0.00$ (the ranking based on pure mutant suspiciousness) localize failure-inducing edits within 19.81 and 17.43 edits, respectively. The reason is that for some failure-inducing edits, accidentally none mapped mutant can simulate their real impacts. Then the suspiciousness values for their mapped mutants can be quite low (even 0.00), making the *Min-Max* strategy and the strategy based on pure mutant suspiciousness perform extremely worse at those cases.

Third, in terms of stability, the *Max-Max* strategy and the *Ratio-Max* strategies with $\alpha \in [0.05, 0.95]$ outperform FAULTTRACER. As Figures 9(a) to 9(d) show, the box plots representing *Max-Max* and *Ratio-Max* strategies with $\alpha \in [0.05, 0.95]$ are consistently more condensed than that of FAULTTRACER based on pure edit suspiciousness. To validate this observation, we also compute the Standard Deviations (SD) for each compared technique. The SDs for the

[7] Note that for each version pair with multiple faulty edits, we use the average ranking of all its faulty edits.

Max-Max strategy and all *Ratio-Max* strategies with α from 0.05 to 0.95 are also consistently smaller than that of FAULTTRACER across all the four formulae.

Fourth, for different formulae, different α values have different impacts for the *Ratio-Max* strategy. For example, all α values perform similarly for both the Ochiai and Jaccard formulae. On the contrary, α values between 0.35 and 0.85 perform better than other values for the SBI formula, and α values between 0.15 and 0.40 perform better than other values for the Tarantula formula. An interesting finding is that even adding a little flavor of mutant suspiciousness to the edit suspiciousness (i.e., α=0.95) would boost the ranking based on pure edit suspiciousness (i.e., FAULTTRACER) significantly. For example, when using the Jaccard formula, in the median case, FAULTTRACER is able to localize faults within 9.5 edits, while the *Ratio-Max* strategy with α=0.95 is able to localize faults within 5.25 edits, thus significantly reducing the burden on developers to localize faults.

Finally, we also perform statistical tests to compare FAULTTRACER with various FIFL strategies[8]. For each FIFL strategy, we use its ranking of failure-inducing edits on different version pairs as a sample data set, and compare it against the corresponding sample set for FAULTTRACER. Before applying paired significance test, we first apply the *Shapiro-Wilk Normality Test* [44] to check the normality assumption. The results show that the differences between any FIFL strategy and FAULTTRACER do not follow normal distribution even at the 0.01 significance level. Therefore, we choose to use the *Wilcoxon Signed-Rank Test* [49] to compare FIFL and FAULTTRACER, because it is suitable for the case that the sample differences may not be normally distributed [29]. Table 4 shows the detailed Wilcoxon test results.

In Table 4, Column 1 shows the various FIFL strategies compared against FAULTTRACER. Columns 2-5 show the p values for comparing the corresponding FIFL strategies with FAULTTRACER when using the four different formulae. The *Null* hypothesis was rejected at the 0.01 significance level (i.e., $p < 0.01$) when comparing all FIFL *Ratio-Max* strategies with $\alpha \in [0.05, 0.95]$ against FAULTTRACER, indicating that the vast majority of FIFL strategies are able to statistically (i.e., not likely to be accidentally) outperform FAULTTRACER in localizing failure-inducing edits. The table also shows that the *Null* hypothesis was not rejected at the 0.01 significance level when comparing FAULTTRACER against FIFL's *Min-Max* strategy, *Max-Max* strategy, and the strategy based on pure mutant suspiciousness (i.e., *Ratio-Max* strategy with α value of 0.00), indicating that these three FIFL strategies may not outperform FAULTTRACER consistently. One interesting finding is that although the FIFL *Max-Max* strategy is able to outperform some FIFL *Ratio-Max* strategy with $\alpha \in [0.05, 0.95]$ in terms of average/median

[8] All the statistical tests used in this paper were performed using the *R* language [21].

FIFL	SBI (p)	Tarantula (p)	Ochiai (p)	Jaccard (p)
Min.	0.1179	0.1179	0.6602	0.2679
Max.	0.1487	0.1147	0.1089	0.1207
R95	0.0006**	0.0011**	0.0011**	0.0006**
R90	0.0006**	0.0007**	0.0007**	0.0007**
R85	0.0006**	0.0011**	0.0004**	0.0007**
R80	0.0006**	0.0009**	0.0012**	0.0006**
R75	0.0006**	0.0008**	0.0006**	0.0010**
R70	0.0007**	0.0008**	0.0006**	0.0011**
R65	0.0006**	0.0008**	0.0006**	0.0013**
R60	0.0006**	0.0012**	0.0007**	0.0013**
R55	0.0004**	0.0006**	0.0004**	0.0008**
R50	0.0004**	0.0004**	0.0005**	0.0009**
R45	0.0004**	0.0004**	0.0006**	0.0008**
R40	0.0004**	0.0004**	0.0005**	0.0017**
R35	0.0004**	0.0004**	0.0008**	0.0010**
R30	0.0013**	0.0004**	0.0009**	0.0035**
R25	0.0013**	0.0004**	0.0021**	0.0035**
R20	0.0013**	0.0004**	0.0024**	0.0035**
R15	0.0014**	0.0004**	0.0038**	0.0040**
R10	0.0014**	0.0013**	0.0040**	0.0040**
R05	0.0014**	0.0012**	0.0045**	0.0040**
R00	0.4665	0.4444	0.4079	0.2762

$*$ indicates significance at the 0.05 level (p<0.05)
$**$ indicates significance at the 0.01 level (p<0.01)

Table 4. Wilcoxon tests for comparing FIFL techniques with FAULTTRACER

performance, the *Max-Max* strategy is not able to outperform FAULTTRACER in terms of significance tests while all FIFL *Ratio-Max* strategies with $\alpha \in [0.05, 0.95]$ outperform FAULTTRACER. The reason is that the performance of the *Max-Max* strategy is not stable for different subjects – it outperforms FAULTTRACER substantially for some subjects but also performs worse than FAULTTRACER for some subjects. On the contrary, although some *Ratio-Max* strategies with $\alpha \in [0.05, 0.95]$ cannot outperform FAULTTRACER substantially, it outperforms FAULTTRACER consistently across different subjects. The results demonstrate that using both edit suspiciousness and mutant suspiciousness for ranking each edit (i.e., FIFL's *Ratio-Max* strategies with $\alpha \in [0.05, 0.95]$) performs better than using either edit suspiciousness or mutant suspiciousness for ranking each edit (i.e., FAULTTRACER that uses pure edit suspiciousness, FIFL *Min-Max* that uses the lower suspiciousness values, FIFL *Max-Max* that uses the higher suspiciousness values, and FIFL *Ratio-Max* with $\alpha = 0.00$ that uses pure mutant suspiciousness). The reason is that both the spectrum information and the impact information (simulated by mutation testing) are useful for localizing failure-inducing edits, and thus using any one of them for one edit may not be both precise and stable.

In summary, both the descriptive statistics and the significance tests show that a vast majority of FIFL strategies are able to outperform FAULTTRACER significantly. Furthermore, the significance tests show that using both edit suspicious-

ness and mutant suspiciousness for ranking each edit performs better than using either edit suspiciousness or mutant suspiciousness for ranking each edit, further demonstrating the motivation of the paper – combining edit spectrum information and edit impact information (simulated by mutation testing) can achieve better fault localization results.

5.4.2 Detailed comparison between FAULTTRACER and FIFL with the default settings

We present the detailed comparison between FAULTTRACER and FIFL's *Ratio-Max* strategy with the default α of 0.50 on all the version pairs. In Table 5, Column 1 lists all the version pairs with regression faults. Columns 2-4 present the average rank of faulty edits by FAULTTRACER, the average rank of faulty edits by FIFL, and improvement by FIFL over FAULTTRACER(%) using the SBI formula for each subject. Note that we also show the improvement achieved by FIFL without mapping approximations for addition edits (Rules 3-9 in Figure 3) in parentheses. Similarly, Columns 5-13 present the comparison between FAULTTRACER and FIFL using the Tarantula, Ochiai, and Jaccard formulae.

In general, using all the formulae, all FIFL techniques in Table 5 are able to achieve improvements over FAULTTRACER for the majority of the version pairs. Also, the statistical test in Table 4 also confirms that all FIFL techniques in Table 5 can statistically outperform FAULTTRACER. For example, using the default SBI formula, FIFL outperforms FAULTTRACER by 2.33% to 86.26% for 16 of 26 version pairs and is only slight inferior than FAULTTRACER on one version pair (with an average improvement of 36.46%). The reason FIFL techniques in Table 5 outperform FAULTTRACER over the vast majority of the studied subjects is that those FIFL techniques use both the edit suspiciousness and mutant suspiciousness for ranking each edit, which provide both coverage and impact information for precise fault localization. The reason FIFL techniques do not outperform FAULTTRACER on every case is that FAULTTRACER techniques already rank the failure-inducing edits precisely using only edit suspiciousness for some version pairs, and the use of mutant suspiciousness may bring some noises to the ranked list. The experimental data supports this reasoning: for example, among the 10 version pairs where FIFL cannot outperform FAULTTRACER using the SBI formula, FAULTTRACER is already able to localize failure-inducing edits within 5 edits for 8 version pairs, leaving little room for FIFL to improve.

We also observe that even FIFL without mapping approximations is also able to outperform FAULTTRACER significantly. For example, using the SBI formula, FIFL without mapping approximations can outperform FAULTTRACER for 14 of 26 version pairs with an average improvement of 30.72%. For some version pairs (e.g., P_{14}), FIFL without mapping approximations even slightly outperforms FIFL with mapping approximations. The reason is that FIFL without mapping approximations aggressively ignores the chance to increase the suspiciousness for addition edits using mutant suspiciousness, and thus performing better for some version pairs with only faults in non-addition edits. However, for the majority of the version pairs, FIFL with mapping approximations performs better.

To further understand the performance of FIFL, we also manually analyzed why FIFL outperform or cannot outperform FAULTTRACER for each subject using the SBI formula. We describe the following interesting cases:

Case 1. When *XStream* evolved from V1.20 to V1.21 (P_{11}), 3 tests failed because the developers added faulty method `XStream.buildMapper()` (shown in Figure 8), which is used to initialize `XStream` object and is executed by every test. Therefore, FAULTTRACER, which treats edits mainly executed by failed tests as more suspicious, cannot rank this `AM` edit high. FIFL without mapping approximations cannot improve over FAULTTRACER because it cannot map the addition edit to any mutant. In contrast, FIFL with mapping approximations maps the edit to mutants inside changed method `XStream.XStream()` using Rules 1 and 9. Some mapped mutants failed exactly the same set of tests with the new program version because those mutants mutate the old statements for building mappers inside `XStream.XStream()`, and therefore boost the ranking of the failure-inducing edit by 76.92%.

Case 2. When *Commons-Lang* evolved from V3.02 to V3.03 (P_{14}), test `FastDateFormatTest.testLang538` failed because the developer removed a conditional block for updating time zone in method `FastDateFormat.format()` (shown in Figure 4). The changed method is used by 35 tests that involve date format transformation. However, only 1 of them failed because the other 34 tests do not check the detailed time zone, making `CM(FastDateFormat.format())` have a suspiciousness value of only 0.0286 using FAULTTRACER, and not able to be ranked high. In contrast, using Rule 1, FIFL maps `CM(FastDateFormat.format())` with 5 mutants (each mapped with a line in the method in V3.02), three of which are killed exactly by the failed test and thus have suspiciousness values of 1.0. In this way, FIFL precisely localizes the failure-inducing program edit within top 2 edits, outperforming FAULTTRACER by 80%. In this case, FIFL without mapping approximation can also localize the fault precisely because the failure-inducing edits is not addition change.

Case 3. When *JMeter* evolved from V1.0 to V2.0, test `testArgumentCreation` in class `ArgumentsPanel.Test` and test `testTreeConversion` in class `Save.Test` failed because of one faulty edit, `AM(NamePanel.updateName())`. Although the suspiciousness value of the edit is already 1.0 using FAULTTRACER, 12 other edits also have the suspiciousness value of 1.0, making FAULTTRACER only rank the failure-inducing edit within top 13 edits. In contrast, FIFL is able to localize the failure-inducing edit within 9 edits because FIFL refines the suspiciousness of each edit based

Rev.	SBI				Tarantula				Ochiai				Jaccard			
	Ft.	Fi.	Improvement(%)		Ft.	Fi.	Improvement(%)		Ft.	Fi.	Improvement(%)		Ft.	Fi.	Improvement(%)	
P_1	4.00	3.00	25.00	(25.00)	4.00	3.00	25.00	(25.00)	4.00	3.00	25.00	(25.00)	4.00	3.00	25.00	(25.00)
P_2	1.00	1.00	0.00	(0.00)	1.00	1.00	0.00	(0.00)	1.00	1.00	0.00	(0.00)	1.00	1.00	0.00	(0.00)
P_3	1.00	1.00	0.00	(0.00)	1.00	1.00	0.00	(0.00)	1.00	1.00	0.00	(0.00)	1.00	1.00	0.00	(0.00)
P_4	6.33	6.33	0.00	(0.00)	6.33	6.33	0.00	(0.00)	2.33	2.33	0.00	(0.00)	2.33	2.33	0.00	(0.00)
P_5	25.00	14.25	43.00	(23.00)	25.00	13.50	46.00	(23.00)	12.25	12.00	2.04	(4.08)	11.00	12.25	-11.36	(-9.09)
P_6	16.00	14.00	12.50	(25.00)	16.00	14.67	8.33	(20.83)	16.00	14.33	10.42	(22.92)	16.00	14.33	10.42	(22.92)
P_7	34.00	32.00	5.88	(2.94)	34.00	32.00	5.88	(2.94)	30.00	16.00	46.67	(23.33)	30.00	16.00	46.67	(23.33)
P_8	16.20	8.40	48.15	(48.15)	16.20	12.20	24.69	(24.69)	14.60	8.40	42.47	(42.47)	16.20	8.40	48.15	(48.15)
P_9	1.00	1.00	0.00	(0.00)	1.00	1.00	0.00	(0.00)	1.00	1.00	0.00	(0.00)	1.00	1.00	0.00	(0.00)
P_{10}	5.50	4.50	18.18	(0.00)	5.50	4.00	27.27	(0.00)	5.00	4.00	20.00	(0.00)	5.00	4.50	10.00	(0.00)
P_{11}	13.00	3.00	76.92	(0.00)	13.00	4.00	69.23	(15.38)	13.00	3.00	76.92	(0.00)	13.00	3.00	76.92	(0.00)
P_{12}	7.00	4.00	42.86	(42.86)	7.00	4.50	35.71	(35.71)	7.00	4.00	42.86	(42.86)	7.00	4.00	42.86	(42.86)
P_{13}	59.45	37.36	37.16	(22.94)	59.45	42.45	28.59	(14.37)	78.73	63.36	19.52	(18.24)	73.64	59.45	19.26	(14.44)
P_{14}	43.67	6.00	86.26	(93.13)	43.67	10.00	77.10	(83.21)	42.00	5.67	86.51	(92.06)	44.00	4.33	90.15	(94.70)
P_{15}	62.86	39.43	37.27	(43.64)	62.86	43.43	30.91	(33.64)	66.00	62.14	5.84	(7.58)	64.86	48.00	25.99	(29.52)
P_{16}	15.40	6.60	57.14	(36.36)	15.40	6.40	58.44	(37.66)	12.20	4.40	63.93	(34.43)	15.40	6.60	57.14	(36.36)
P_{17}	4.00	4.00	0.00	(0.00)	4.00	4.00	0.00	(0.00)	4.00	4.00	0.00	(0.00)	4.00	4.00	0.00	(0.00)
P_{18}	13.00	9.00	30.77	(15.38)	13.00	9.00	30.77	(15.38)	13.00	9.00	30.77	(15.38)	13.00	9.00	30.77	(15.38)
P_{19}	32.25	31.50	2.33	(2.33)	32.25	31.00	3.88	(3.10)	48.50	47.00	3.09	(2.58)	51.00	49.75	2.45	(4.41)
P_{20}	9.00	9.00	0.00	(0.00)	9.00	9.00	0.00	(0.00)	9.00	9.00	0.00	(0.00)	9.00	9.00	0.00	(0.00)
P_{21}	10.00	2.00	80.00	(80.00)	10.00	2.00	80.00	(80.00)	10.00	2.00	80.00	(80.00)	10.00	2.00	80.00	(80.00)
P_{22}	2.50	2.50	0.00	(0.00)	2.50	3.00	-20.00	(-20.00)	2.50	2.50	0.00	(0.00)	2.50	2.50	0.00	(0.00)
P_{23}	1.50	1.50	0.00	(0.00)	1.50	1.50	0.00	(0.00)	1.50	1.50	0.00	(0.00)	1.50	1.50	0.00	(0.00)
P_{24}	3.00	3.00	0.00	(0.00)	3.00	3.00	0.00	(0.00)	2.50	2.50	0.00	(0.00)	2.50	2.50	0.00	(0.00)
P_{25}	10.00	6.00	40.00	(30.00)	10.00	6.00	40.00	(30.00)	10.00	6.00	40.00	(30.00)	10.00	6.00	40.00	(30.00)
P_{26}	3.67	4.00	-9.09	(-18.18)	3.67	3.67	0.00	(-9.09)	2.00	2.67	-33.33	(-33.33)	3.67	3.67	0.00	(-9.09)
Avg	15.40	9.78	36.46	(30.72)	15.40	10.45	32.14	(26.12)	15.74	11.22	28.67	(23.75)	15.87	10.74	32.35	(27.49)

Table 5. Comparison between FAULTTRACER and default settings of FIFL.

on their mapping mutants, and decreases the suspiciousness values of 4 top-ranked fault-free edits.

Case 4. When *Mime4J* evolved from V0.61 to V0.70, 3 tests failed because of 4 failure-inducing program edits. FAULTTRACER and FIFL perform similarly on 2 failure-inducing edits. The other 2 failure-inducing edits are `CM` and `AM` edits on constructors of `MimeBoundaryInputStream` class. Because the 2 failure-inducing edits are constructor edits, they are executed by almost all tests, making FAULTTRACER only localize them within 25 edits. In contrast, FIFL maps the two constructor edits with mutants using Rule 1 and Rule 3, respectively. Some mapped mutants have the suspiciousness value of 1.0 because they are only killed by one failed test, making FIFL able to localize the two failure-inducing edits within top 2 edits.

Case 5. When *Xml-Security* evolved from V1.0 to V2.0 (P_{10}), there are two failure-inducing edits: `AM` on method `engineCanonicalizeXPathNodeSet()` inside the class `CanonicalizerBase`, and `CM` on `circumventBug2650()` in class `XMLUtils`. Using Rule 7, FIFL finds that mutants in the deleted method `engineCanonicalizeXPathNodeSet()` of class `Canonicalizer20010315` are able to simulate the impacts of the newly added method. The mapped mutants increase the rank of the failure-inducing `AM` edit by 1. In addition, `AF(CanonicalizerBase._includeComments)` can also be mapped with mutants inside a deleted method by applying Rules 7 and 9. Then, `AF` edit's rank was lowered correspondingly, making the rank of the failure-inducing `CM` edit, `CM(XMLUtils.circumventBug2650())`, increased by 1. Therefore, the average ranking for two failure-inducing edits is improved by 18.18%.

Case 6. The evolution from *Joda-Time*1.20 to 1.30 (P_{26}) is the only case where FAULTTRACER outperforms FIFL using SBI. The reason is that one failure-inducing edit, `CSFI(GregorianChronology.MAX_YEAR)`, does not have mapped mutants, and another fault-free edit, `CM(BasicChronology.getYear())`, has mapped mutants that accidentally share some failed tests with the new version.

In summary, the *Ratio-Max* strategy of FIFL with default setting is able to outperform FAULTTRACER significantly. For example, even the default setting of FIFL with SBI formula outperforms FAULTTRACER by 2.33% to 86.26% on 16 of 26 studied version pairs, and is only inferior than FAULTTRACER on one version pair.

5.4.3 Discussions

Although automated fault localization approaches [1, 16, 23, 28, 38, 52–54, 57, 65] have been intensively studied for more than a decade, there are common limitations for them. For example, Parnin and Orso [37] recently argued that existing fault localization approaches rely on a strong assumption that examining a potential faulty statement in isolation is enough to localize and fix a fault. They performed a case study showing that a traditional fault localization technique (which ranks all program statements to localize faults) does not help the developer much with localizing faults. The study shows that traditional fault localization at the statement granularity can be painful because (1) it may cause the developer to inspect a extremely long ranked list for large program and (2) developers tend to also inspect the context of each statement other than the statement itself. Therefore, the study suggested that fault localization at the method or file granulari-

R50 Formula	Rank all edits			Rank edits per test		
	Ft.	Fi.	Impr.	Ft.	Fi.	Impr.
SBI	15.40	9.78	36.46%	10.12	5.83	42.44%
Tarantula	15.40	10.45	32.14%	10.12	6.10	39.70%
Ochiai	15.74	11.22	28.67%	10.43	6.21	40.48%
Jaccard	15.87	10.74	32.35%	10.80	6.00	44.47%

Table 6. Summary results when using the default *R50* strategy to rank all edits and rank edits for each failed test

ties may be a promising direction for fault localization, because those granularities provide a shorter candidate list and provide enough context information for each ranked entity. Although our FIFL approach may share the same limitations with traditional fault localization, FIFL makes attempts to address the limitations of traditional fault localization. For example, FIFL focuses on program edits during software evolution, which provides a much shorter ranked list than ranking all program statements. In addition, FIFL extract program edits at the method/field level, which provides the developer enough context information for reasoning each ranked entity.

There is also another intrinsic limitation for the spectrum-based fault localization approaches that FIFL is based on [1, 23, 28, 52, 57] – they only use the correlation between the coverage of program elements with test pass/fail results to localize faults. However, there is a gap between the coverage and the actual impact of program elements to the test pass/fail results. In this paper, we make an attempt to bridge the gap by using the simulated impact information via mutation testing. However, the impact information simulated by mutation testing may be still not precise enough. In addition, some edits may not even have mapped mutants due to various reasons, e.g., the mutation operators do not support the specific statement pattern. In the future, we hope more research efforts can be put into this area to bridge the gap between program coverage information and actual impact information using more advanced techniques.

Our experimental evaluation also has limitations. In this paper, we used FAULTTRACER and FIFL to directly rank all the edits once for all failed tests. This corresponds to the debugging process that the developer iterates over all the edits to find all the potential faults for the failed program version. However, some deveopers may prefer to inspect related edits for each failed test to fix failed tests one by one. Therefore, we also used FAULTTRACER and FIFL to rank edits related to each failed test (i.e., only ranking the edits that are affecting changes of each failed test based on their suspiciousness). In this case, we would have a ranked list of edits for each failed test. For each subject, we collected the average rank of each failure-inducing edit on each failed test. Table 6 shows the average summary results across all subjects for FAULTTRACER and FIFL's default strategy when ranking all edits together and when ranking edits for each failed test. In the table, Column 1 presents the four formulae. Columns 2-4 show the average rank of failure-inducing edits by FAULTTRACER and FIFL when ranking all edits together, and the improvement of FIFL over FAULTTRACER. Similarly, Columns 5-7 present the comparison between FIFL and FAULTTRACER when ranking edits for each failed test. We can observe that FIFL outperforms FAULTTRACER even more when ranking edits for each failed test. For example, when using the default strategy with the SBI formula, FIFL is able to localize failure-inducing edits within 5.83 edits for each failed test, indicating an improvement of more than 40% over FAULTTRACER.

Last but not least, Murphy-Hill et al. [33] recently presented an interesting study showing a suite of factors that can cause a program fault to be fixed in different ways at different circumstances or time points. One such factor is the development phase of the project. For example , when fixing a fault at an earlier phase, developers may choose to fix the root cause of the fault so that if a risk raises, they would have a longer period to compensate. On the contrary, when fixing a fault at a later phase, developers may choose to make a walk-around which would be "least disruptive". The findings in this study raises serious questions for traditional bug prediction [24, 66] or fault localization techniques [1, 23, 28, 52, 57], because faults can actually be fixed in different ways and locations rather than the root causes. The effectiveness evaluation of FIFL may also be influenced, because we only evaluate FIFL in localizing the root cause of failure-inducing edits, whereas the developer may choose to fix the faults in different ways at different program locations. However, we believe that FIFL may still be useful for developers even when they finally decide not to fix the faults at the root-cause locations, because fully understanding of the fault root cause is still preferred no matter where the faults are finally fixed (also confirmed by the same study [33]).

5.5 Threats to Validity

Threats to internal validity are concerned with uncontrolled factors that may also be responsible for the results. In this paper, the main threat to internal validity is the possible faults in the implementation of the compared techniques. To reduce this threat, we built FIFL on top of state-of-the-art tools [42, 57], and implemented FIFL using well-known libraries such as *Eclipse JDT toolkit* and *ASM bytecode manipulation framework*. We also reviewed all the code that we produced to assure its correctness. The first author, with Java programming experience for eight years, isolated the failure-inducing edits manually. We have also reviewed all outputs produced by FIFL manually to ensure correctness. However, because this inspection was done manually, there is still a risk of introducing subjectivity and errors.

Threats to external validity are concerned with whether the findings in our study are generalizable for other situations. To mitigate threats to external validity, we used all released versions of nine medium sized open source projects from various application areas. In addition, as our work is related to both regression testing and mutation testing, we also ensure that the selected subjects have been used for regres-

sion testing or mutation testing research [6, 7, 10, 31, 42, 43, 57–59, 61, 63]. However, they still might be not representative for all the possible subject programs.

Threats to construct validity are concerned with whether the measurement in our study reflects real-world situations. To mitigate threats to construct validity, we measured the ranking of failure-inducing edits, which denotes the number of edits that the developer need to manually inspect before finding the fault. Furthermore, we also compare FIFL with the existing technique for localizing failure-inducing edits (FAULTTRACER [57]) in the same experimental setting. The ranking of failure-inducing program elements has been widely used in the fault localization research area [1, 23, 37, 39, 40, 52, 57]. However, the ranking of failure-inducing program elements may still not correlate with the actual costs in inspecting those elements. To further reduce this threat, inspired by user case studies in other areas [25, 30], rigorous and well-designed studies for investigating the correlation between the ranking of faulty elements and actual fault localization costs should be performed in future work. In addition, as recent work has shown that it is suitable to use program repair techniques to evaluate fault localization techniques fully automatically [38], we also plan to use automated program repair techniques [27, 48] to further demonstrate the effectiveness of FIFL.

6. Related Work

As our approach aims to localize failure-inducing program edits precisely, we mainly discuss the existing efforts for this purpose in this section. We also show the related work in traditional fault localization, which aims to localize faulty program elements for a specific program version. In addition, as our approach utilizes the mutation testing methodology for improving failure-inducing edit localization, we also briefly discuss recent trends on mutation testing and its applications.

6.1 Localizing Failure-Inducing Edits

Many research efforts have been dedicated to localizing failure-inducing program edits. *Change impact analysis* is a well-known methodology for determining affecting changes, i.e., a subset of program edits that might have caused test failures, for each failed test. It has been shown that the number of affecting changes for each failed test can still be large in number [57]. Therefore, various techniques have been proposed to localize failure-inducing changes more precisely. Crisp provided tool supports for manually isolating failure-inducing program edits [5]. Stoerzer et al. [45] proposed to use change classification techniques to find failure-inducing changes. However, those techniques do not rank changes and the number of classified changes can still be large in number. Ren et al. [39] proposed the first ranking heuristic for localizing failure-inducing edits. However, the ranking heuristic is only based on calling structures of failed tests, and can only rank method-level edits. Recent work [2, 57, 62] (e.g. FAULTTRACER) introduced spectrum-based fault localization methodology to localize failure-inducing edits. FAULTTRACER can rank all types of edits, and has been shown to outperform Ren et al.'s ranking heuristic by more than 50%. However, some edits ranked as top by FAULTTRACER might just because they are accidentally executed by the failed tests. In fact, they might not be responsible for test failures.

A natural idea to further refine the fault localization results would be iteratively applying subsets of program edits to localize the failure-inducing edits precisely. However, due to the limitations for applying edits automatically (details are shown in Section 2), existing techniques for localizing failure-inducing edits usually recommend manually applying changes after ranking the edits [5, 39, 57]. In contrast, FIFL directly uses the existing mutation testing results of the old program version to boost the fault localization results while avoiding the limitations of iteratively applying subsets of program edits. Note that similar with existing techniques [5, 39, 57], FIFL can also be combined with manually applying program edits to further facilitate fault localization.

Delta Debugging [32, 51, 53, 54] is another methodology for localizing failure-inducing edits. Delta Debugging iteratively applies a subset of all changes to construct intermediate versions to find a minimum set of changes that lead to a test failure. However, Delta Debugging considers all changes as the candidate set without considering compilation dependences among those changes, and needs costly test execution for validating various combinations of program changes. Also, Delta Debugging does not work for multiple thread programs as shown by previous work [46] because it always assumes that tests give deterministic results. Furthermore, Delta Debugging does not rank edits, leaving it to a programmer to sort out a real culprit of a regression test failure.

To the best of our knowledge, our FIFL approach is the first approach to localizing failure-inducing edits based on mutation testing. FIFL unifies two dimensions of changes to consider the potential impact of each program edit. The consideration of both spectra and impacts of program edits makes FIFL more precise than the existing techniques.

6.2 Traditional Fault Localization

Traditional fault localization techniques [1, 15, 17, 22, 23, 28, 36, 52, 64] aims to localize faults among all the elements of a specific program version, and may suffer from scalability issues when applied to large evolving software systems, since they do not leverage edits between the old and new versions. A number of spectrum-based techniques [1, 15, 23, 28, 52] have been proposed to localize faults by analyzing the correlation between test fail/pass results and test coverage information. Due to the imprecision of the spectrum-based techniques, Zhang et al. [64] proposed the *predicate switching* approach, which forces critical predicates to take an alternate branch at runtime. By examining the switched predicate that caused a failed test to pass, the cause of the

bug then may be identified. Jeffrey et al. [22] further proposed the *value replacement* approach, which subsumes the *predicate switching* approach and forces variables at each program location to take different values at runtime to localize potential faulty statements. Zhang et al. [63] then proposed to replace test objects using randomly constructed object pool for program test code to explain the potential causes of test failures. A recent work by Papadakis et al. [36] is closely related to our work. Their work to our knowledge is the first to utilize mutation testing to facilitate traditional fault localization. Our FIFL differs from their work in three key ways. First, their work only considers mutation changes and aims to localize faults for one specific version, whereas FIFL considers two dimensions of changes, including mutation changes and programmer edits, and aims to localize faulty edits during software evolution. Second, our approach is different: their approach uses mutation testing as a coverage criterion, whereas FIFL uses mutation testing to simulate the impact of program edits. Third, their technique is not applicable to tests with assertions, which are widely used in real-world systems. The reason is that they directly apply mutation testing on the faulty program with failed tests and the set of mutants killed by already failed tests cannot be determined. In contrast, FIFL applies to more general forms of tests because FIFL applies mutation testing on the old version where all tests pass.

6.3 Mutation Testing

Mutation testing, first proposed by DeMillo [9] and Hamlet [14], is a fault-based testing methodology. Mutation testing has been shown to be effective in indicating the quality of test suites, and is raising more and more attention from both the industry and the academia. As mutation testing is widely recognized as a heavy-weight testing methodology, many researchers aim to reduce the cost of mutation testing. Selective mutation testing techniques [13, 34, 55] select a representative subset of all mutants and ensure that the selected set of mutants achieves almost the same results as the whole mutant set. Howden [20] proposed weak mutation, which checks whether the test could result in a different internal state in the mutant from that in the original program rather than check the final results. Untch et al. [47] proposed schema-based mutation, which transforms all mutants into one meta-mutant that can be compiled by a standard compiler. Researchers have also investigated the use of parallel processing (e.g., SIMD [26]) to speed up mutation testing. Recently, Zhang et al. [58, 60] proposed techniques inspired by regression testing to further reduce mutation testing cost.

Mutation testing has also been traditionally used for generating high-quality test suites. DeMillo and Offutt [8] proposed constraint based testing (CBT), which uses control-flow analysis and symbolic evaluation to generate tests each killing one mutant. Offutt et al. [35] further proposed dynamic domain reduction (DDR) to address some limitations of CBT. Fraser and Zeller [12] proposed to use search based software testing (SBST) to generate tests each killing one mutant. Zhang et al. [56] proposed to use dynamic symbolic execution (DSE) to generate tests each killing one mutant. Harman et al. [18] proposed to combine DSE and SBST to generate tests each killing multiple mutants.

Different from traditional applications of mutation testing, our FIFL unifies two dimensions of changes – mutation changes and developer edits – to localize failure-inducing program edits more precisely.

7. Conclusion

This paper unifies two widely studied dimensions of software changes – mutation changes [4, 9, 12, 14, 18, 20, 34, 42, 43, 56] and developer edits [5, 39, 40, 45, 50, 51, 57] – and presents our methodology of fault injection for fault localization (FIFL). Our insight is that the essence of failure-inducing edits made by the developer can be captured using mechanical program transformations (i.e., mutation changes in this paper). Specifically, mutants of the old program version, which have similar test execution results as the new version, are likely to share the same locations with some failure-inducing edits. As an optimization to reduce the cost of FIFL, we also present *edit-oriented mutation testing*, which only executes the subset of mutants mapped with program edits when the mutation testing results for the old version are not available when applying FIFL. We performed an empirical study on the real-world versions of nine open-source Java projects. The empirical results show that the FIFL approach achieves considerable improvement (i.e., even up to more than 80% for some subjects under test) over state-of-the-art FAULTTRACER approach.

The unified view of mutation changes and developer edits is not limited to the fault localization area. We believe it is also applicable to other key software testing areas. For example, mutation testing results for the old program version can optimize test selection [19] and prioritization [41] for the new version, because the potential impact of program edits can be simulated by existing mutants. We plan to establish these connections in future work.

Acknowledgments

We thank the anonymous reviewers for the valuable reviews and suggestions that helped improving the paper a lot during the second-phase revision. This material is based upon work partially supported by the US National Science Foundation under Grant No. CCF-0845628. Lu Zhang's work is sponsored by the National 973 Program of China No. 2011CB302604, the National 863 Program of China No. 2012AA011202, the Science Fund for Creative Research Groups of China No. 61121063, and the National Natural Science Foundation of China under Grant Nos. 91118004, 61225007, 61228203.

References

[1] R. Abreu, P. Zoeteweij, and A. J. Van Gemund. On the accuracy of spectrum-based fault localization. In *Testing: Academic and Industrial Conference Practice and Research Techniques-MUTATION*, pages 89–98, 2007.

[2] E. Alves, M. Gligoric, V. Jagannath, and M. d'Amorim. Fault-localization using dynamic slicing and change impact analysis. In *Proc. of ASE*, pages 520–523, 2011.

[3] P. Ammann and J. Offutt. *Introduction to software testing*. Cambridge University Press, 2008.

[4] T. A. Budd, R. A. DeMillo, R. J. Lipton, and F. G. Sayward. Theoretical and empirical studies on using program mutation to test the functional correctness of programs. In *Proc. of POPL*, pages 220–233, 1980.

[5] O. Chesley, X. Ren, B. Ryder, and F. Tip. Crisp–A Fault Localization Tool for Java Programs. In *Proc. of ICSE*, pages 775–779, 2007.

[6] B. Daniel, V. Jagannath, D. Dig, and D. Marinov. Reassert: Suggesting repairs for broken unit tests. In *Proc. of ASE*, pages 433–444, 2009.

[7] B. Daniel, T. Gvero, and D. Marinov. On test repair using symbolic execution. In *Proc. of ISSTA*, pages 207–218, 2010.

[8] R. DeMillo and A. Offutt. Constraint-based automatic test data generation. *IEEE Transactions on Software Engineering*, 17(9):900–910, 1991.

[9] R. DeMillo, R. Lipton, and F. Sayward. Hints on test data selection: Help for the practicing programmer. *Computer*, 11 (4):34–41, 1978.

[10] H. Do, S. Elbaum, and G. Rothermel. Supporting controlled experimentation with testing techniques: An infrastructure and its potential impact. *Empirical Software Engineering*, 10 (4):405–435, 2005. ISSN 1382-3256.

[11] P. G. Frankl, S. N. Weiss, and C. Hu. All-uses vs mutation testing: an experimental comparison of effectiveness. *Journal of Systems and Software*, 38(3):235–253, 1997.

[12] G. Fraser and A. Zeller. Mutation-driven generation of unit tests and oracles. In *Proc. of ISSTA*, pages 147–158, 2010.

[13] M. Gligoric, L. Zhang, C. Pereira, and G. Pokam. Selective mutation testing for concurrent code. In *Proc. of ISSTA*, pages 224–234, 2013.

[14] R. Hamlet. Testing programs with the aid of a compiler. *IEEE Transactions on Software Engineering*, (4):279–290, 1977.

[15] D. Hao, L. Zhang, Y. Pan, H. Mei, and J. Sun. On similarity-awareness in testing-based fault localization. *Automated Software Engineering*, 15(2):207–249, 2008.

[16] D. Hao, L. Zhang, T. Xie, H. Mei, and J.-S. Sun. Interactive fault localization using test information. *Journal of Computer Science and Technology*, 24(5):962–974, 2009.

[17] D. Hao, T. Xie, L. Zhang, X. Wang, J. Sun, and H. Mei. Test input reduction for result inspection to facilitate fault localization. *Automated Software Engineering*, 17(1):5–31, 2010.

[18] M. Harman, Y. Jia, and W. Langdon. Strong higher order mutation-based test data generation. In *Proc. of FSE*, pages 212–222, 2011.

[19] 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 *Proc. of OOPSLA*, pages 312–326, 2001.

[20] W. Howden. Weak mutation testing and completeness of test sets. *IEEE Transactions on Software Engineering*, (4):371–379, 1982.

[21] R. Ihaka and R. Gentleman. R: A language for data analysis and graphics. *Journal of computational and graphical statistics*, 5(3):299–314, 1996.

[22] D. Jeffrey, N. Gupta, and R. Gupta. Fault localization using value replacement. In *Proc. of ISSTA*, pages 167–178, 2008.

[23] J. Jones, M. Harrold, and J. Stasko. Visualization of test information to assist fault localization. In *Proc. of ICSE*, page 477, 2002.

[24] S. Kim, E. J. Whitehead, and Y. Zhang. Classifying software changes: Clean or buggy? *IEEE TSE*, 34(2):181–196, 2008.

[25] 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 TSE*, 32(12):971–987, 2006.

[26] E. Krauser, A. Mathur, and V. Rego. High performance software testing on simd machines. *IEEE Transactions on Software Engineering*, 17(5):403–423, 1991.

[27] C. Le Goues, M. Dewey-Vogt, S. Forrest, and W. Weimer. A systematic study of automated program repair: Fixing 55 out of 105 bugs for $8 each. In *Proc. of ICSE*, pages 3–13, 2012.

[28] B. Liblit, M. Naik, A. Zheng, A. Aiken, and M. Jordan. Scalable statistical bug isolation. In *Proc. of PLDI*, pages 15–26, 2005. ISBN 1595930566.

[29] R. Lowry. *Concepts and applications of inferential statistics*. R. Lowry, 1998.

[30] C. Mayer, S. Hanenberg, R. Robbes, É. Tanter, and A. Stefik. An empirical study of the influence of static type systems on the usability of undocumented software. In *Proc. of OOPSLA*, pages 683–702, 2012.

[31] H. Mei, D. Hao, L. Zhang, L. Zhang, J. Zhou, and G. Rothermel. A static approach to prioritizing junit test cases. *TSE*, 38 (6):1258–1275, 2012.

[32] G. Misherghi and Z. Su. Hdd: hierarchical delta debugging. In *Proc. of ICSE*, pages 142–151, 2006.

[33] E. Murphy-Hill, T. Zimmermann, C. Bird, and N. Nagappan. The design of bug fixes. In *Proc. of ICSE*, pages 332–341, 2013.

[34] A. Offutt, A. Lee, G. Rothermel, R. Untch, and C. Zapf. An experimental determination of sufficient mutant operators. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 5(2):99–118, 1996.

[35] A. Offutt, Z. Jin, and J. Pan. The dynamic domain reduction procedure for test data generation. *Software-Practice and Experience*, 29(2):167–194, 1999.

[36] M. Papadakis and Y. L. Traon. Using mutants to locate unknown faults. In *Proc. of ICST Workshop on Mutation Analysis*, pages 691–700, 2012.

[37] C. Parnin and A. Orso. Are automated debugging techniques actually helping programmers? In *Proc. of ISSTA*, pages 199–209, 2011.

[38] Y. Qi, X. Mao, Y. Lei, and C. Wang. Using automated program repair for evaluating the effectiveness of fault localization techniques. In *Proc. of ISSTA*, pages 191–201, 2013.

[39] X. Ren and B. Ryder. Heuristic ranking of Java program edits for fault localization. In *Proc. of ISSTA*, pages 239–249, 2007.

[40] X. Ren, F. Shah, F. Tip, B. Ryder, and O. Chesley. Chianti: A tool for change impact analysis of Java programs. In *Proc. of OOPSLA*, 2004.

[41] G. Rothermel, R. H. Untch, C. Chu, and M. J. Harrold. Prioritizing test cases for regression testing. *IEEE Transactions on Software Engineering*, 27(10):929–948, 2001.

[42] D. Schuler and A. Zeller. Javalanche: Efficient mutation testing for Java. In *Proc. of FSE*, pages 297–298, 2009.

[43] D. Schuler, V. Dallmeier, and A. Zeller. Efficient mutation testing by checking invariant violations. In *Proc. of ISSTA*, pages 69–80, 2009.

[44] S. S. Shapiro and M. B. Wilk. An analysis of variance test for normality (complete samples). *Biometrika*, 52(3/4):591–611, 1965.

[45] M. Stoerzer, B. Ryder, X. Ren, and F. Tip. Finding failure-inducing changes in Java programs using change classification. In *Proc. of FSE*, pages 57–68, 2006.

[46] R. Tzoref, S. Ur, and E. Yom-Tov. Instrumenting where it hurts: an automatic concurrent debugging technique. In *Proc. of ISSTA*, pages 27–38, 2007.

[47] R. Untch, A. Offutt, and M. Harrold. Mutation analysis using mutant schemata. In *ACM SIGSOFT Software Engineering Notes*, volume 18, pages 139–148, 1993.

[48] W. Weimer, T. Nguyen, C. Le Goues, and S. Forrest. Automatically finding patches using genetic programming. In *Proc. of ICSE*, pages 364–374, 2009.

[49] F. Wilcoxon. Individual comparisons by ranking methods. *Biometrics bulletin*, 1(6):80–83, 1945.

[50] G. Xu and A. Rountev. Regression test selection for aspectj software. In *Proc. of ICSE*, pages 65–74, 2007.

[51] K. Yu, M. Lin, J. Chen, and X. Zhang. Practical isolation of failure-inducing changes for debugging regression faults. In *Proc. of ASE*, pages 20–29, 2012.

[52] Y. Yu, J. Jones, and M. Harrold. An empirical study of the effects of test-suite reduction on fault localization. In *Proc. of ICSE*, pages 201–210, 2008.

[53] A. Zeller. Yesterday, my program worked. today, it does not. why? In *Proc. of FSE*, pages 253–267, 1999.

[54] A. Zeller. Automated debugging: Are we close? *Computer*, 34(11):26–31, 2001.

[55] L. Zhang, S.-S. Hou, J.-J. Hu, T. Xie, and H. Mei. Is operator-based mutant selection superior to random mutant selection? In *Proc. of ICSE*, pages 435–444, 2010.

[56] L. Zhang, T. Xie, L. Zhang, N. Tillmann, J. De Halleux, and H. Mei. Test generation via dynamic symbolic execution for mutation testing. In *Proc. of ICSM*, pages 1–10, 2010.

[57] L. Zhang, M. Kim, and S. Khurshid. Localizing failure-inducing program edits based on spectrum information. In *Proc. of ICSM*, pages 23–32, 2011.

[58] L. Zhang, D. Marinov, L. Zhang, and S. Khurshid. Regression mutation testing. In *Proc. of ISSTA*, pages 331–341, 2012.

[59] L. Zhang, D. Hao, L. Zhang, G. Rothermel, and H. Mei. Bridging the gap between the total and additional test-case prioritization strategies. In *Proc. ICSE*, pages 192–201, 2013.

[60] L. Zhang, D. Marinov, and S. Khurshid. Faster mutation testing inspired by test prioritization and reduction. In *Proc. of ISSTA*, pages 235–245, 2013.

[61] S. Zhang. Practical semantic test simplification. In *Proc. of ICSE*, pages 1173–1176, 2013.

[62] S. Zhang, Y. Lin, Z. Gu, and J. Zhao. Effective identification of failure-inducing changes: a hybrid approach. In *Proc. of PASTE*, pages 77–83, 2008.

[63] S. Zhang, C. Zhang, and M. D. Ernst. Automated documentation inference to explain failed tests. In *Proc. of ASE*, pages 63–72, 2011.

[64] X. Zhang, N. Gupta, and R. Gupta. Locating faults through automated predicate switching. In *Proc. of ICSE*, pages 272–281, 2006.

[65] X. Zhang, N. Gupta, and R. Gupta. Pruning dynamic slices with confidence. In *Proc. of PLDI*, pages 169–180, 2006.

[66] T. Zimmermann, N. Nagappan, P. J. Guo, and B. Murphy. Characterizing and predicting which bugs get reopened. In *Proc. of ICSE*, pages 1074–1083. IEEE, 2012.