Understanding Regression Failures through Test-Passing and Test-Failing Code Changes

Roykrong Sukkerd[†], Ivan Beschastnikh[†], Jochen Wuttke[†], Sai Zhang[†], Yuriy Brun[UM]
[†]University of Washington
Seattle, WA, USA
{rsukkerd, ivan, wuttke, szhang}@cs.washington.edu
[UM]University of Massachusetts
Amherst, MA, USA
brun@cs.umass.edu

***Abstract*—Debugging and isolating changes responsible for regression test failures are some of the most challenging aspects of modern software development. Automatic bug localization techniques reduce the manual effort developers spend examining code, for example, by focusing attention on the minimal subset of recent changes that results in the test failure, or on changes to components with most dependencies or highest churn.**

We observe that another subset of changes is worth the developers' attention: the complement of the maximal set of changes that does not produce the failure. While for simple, independent source-code changes, existing techniques localize the failure cause to a small subset of those changes, we find that when changes interact, the failure cause is often in our proposed subset and not in the subset existing techniques identify.

In studying 45 regression failures in a large, open-source project, we find that for 87% of those failures, the complement of the maximal passing set of changes is different from the minimal failing set of changes, and that for 78% of the failures, our technique identifies relevant changes ignored by existing work. These preliminary results suggest that combining our ideas with existing techniques, as opposed to using either in isolation, can improve the effectiveness of bug localization tools.

I. Introduction

When a regression test fails, developers examine the changes since the last time the test passed to identify the failure cause.[2] Considering the minimal subsets of changes that break the test can help developers localize the failure cause and identify flaws in their reasoning [5], [10], [11].

Intuitively, changes made since the last time the test passed can be divided into two subsets: the minimal set of changes that produces the failure (we call this set Δ_f) and the maximal set of changes that does not produce the failure (we call this set Δ_p). However, interactions between the changes may violate this intuition, resulting in more complex relationships than $\Delta_f = \overline{\Delta_p}$. In practice, we find that for 87% of regression failures, these sets are not complementary (see Section III). Thus, $\overline{\Delta_p}$ may contain information relevant to debugging that is not in Δ_f, and that information may be used to improve existing techniques that focus the developer on parts of Δ_f or efficiently computed approximations of Δ_f [10].

To understand why Δ_f and Δ_p may not be complementary, consider an example: A developer introduces two changes, each of which independently causes a test failure. The Δ_f (what existing techniques find) consists of only a single change. Meanwhile both of the changes are in $\overline{\Delta_p}$ (the complement of Δ_p). Further, consider a developer who is given Δ_f. If she were to fix the problem in Δ_f, the test would still fail. This can be misleading and can make debugging unnecessarily hard. Here, $\overline{\Delta_p}$ is more relevant to the test failure. Other common situations, such as writing a buggy method that is not exercised by the regression tests until another change calls that method, can cause even more headaches. Overall, for 78% of the regression failures we examined, $\overline{\Delta_p}$ contained changes that were not in Δ_f.

The main contribution of this paper is the counterintuitive observation that Δ_f and Δ_p are often not complementary. We examine the possible relationships between Δ_f and $\overline{\Delta_p}$, and provide source-code examples to illustrate how some of these unexpected situations may occur in practice. Further, in analyzing the revision history of an open-source system, we find that the counterintuitive relationships between Δ_f and $\overline{\Delta_p}$ occur frequently: 87% of the time. **The impact of this work is that this observation can lead to improving the effectiveness of existing techniques**, such as delta debugging. In our preliminary study, for 78% of the 45 real regression failures we examined, $\overline{\Delta_p}$ provides relevant information ignored by techniques that only consider Δ_f.

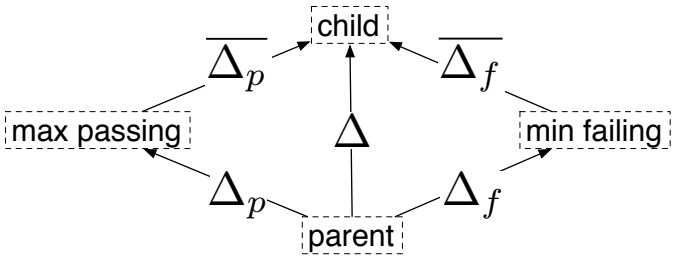

Fig. 1. A conceptual overview of the relationship between $\overline{\Delta_p}$ and Δ_f, two sets of changes we aim to use to help understand and localize bugs.

[1]This work is supported by NSF grants CNS-0855252 and CCF-0963757, and DARPA contracts FA8750-12-2-0107 and FA8750-12-C-0174.

[2]We refer to the parts of the source code that cause a test to fail as the *failure cause*. While the *defect*, which may be in the tests, in the requirements, or elsewhere in the project, is the real cause of the problem, the failure cause is often the defect and developers often attempt to find it first.

978-1-4673-3076-3/13/$31.00 © 2013 IEEE

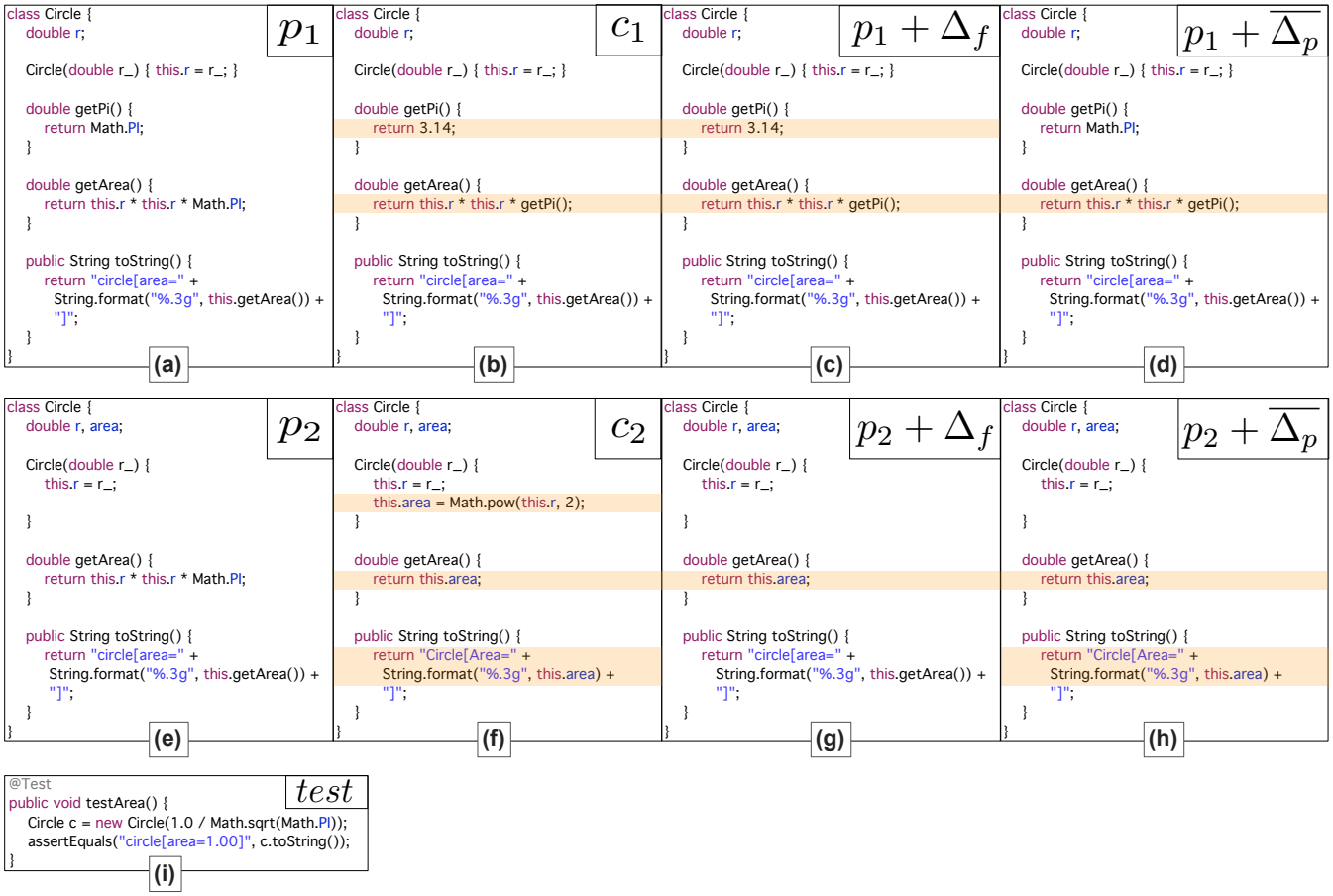

Fig. 2. Examples of two changes to a `Circle` class that break the test in **(i)**. In the example in **(a)–(d)**, the failure cause lies in Δ_f but not in $\overline{\Delta_p}$ (case ③ in Figure 3). In the example in **(e)–(h)**, the failure cause lies in $\overline{\Delta_p}$ but not in Δ_f, and also lies outside of both sets (case ④ in Figure 3).

II. Unpacking Δ: Interesting Change Subsets

Consider a regression test t and a *parent* (p) version of a codebase that passes t, along with a set of changes Δ that when applied to p produces the *child* (c) version that fails t. Figure 1 overviews the relationships between p, c and the five sets of changes this section defines: Δ, Δ_p, $\overline{\Delta_p}$, Δ_f, and $\overline{\Delta_f}$. For simplicity, we assume here that Δ does not modify the source of t, and we do not consider external causes of t's failure, which are discussed elsewhere [9].

Our goal in localizing a bug is to identify a small subset of changes that reproduces t's failure in the same way as the entire set of changes. Existing bug localization work that considers the changes' impact on t focuses on finding the minimal subset of Δ that, when applied to p, is compilable and reproduces t's failure [5], [10], [11]. We call this subset Δ_f, and refer to p with Δ_f applied as the "*minimal failing*" version. Note that there always exists at least one non-empty Δ_f (Δ_f may equal Δ), and there may be multiple, equal-sized Δ_f.

Another relevant subset of Δ is the maximal subset of Δ that, when applied to p, is compilable and passes t. In other words, all the changes in Δ that do not break the test. We call this subset Δ_p, and refer to p with Δ_p applied as the "*maximal passing*" version. Note that Δ_p must be a proper subset of Δ, Δ_p may be empty, and there may be multiple, equal-sized Δ_p.

Since Δ_f captures just those changes that cause the test to fail, intuitively, Δ_f and Δ_p should be each other's complements: $\Delta_p \cup \Delta_f = \Delta$ and $\Delta_p \cap \Delta_f = \emptyset$. In other words, Δ_f should be equal to $\overline{\Delta_p} = \Delta \setminus \Delta_p$, the complement of Δ_p. However, in practice, we find that often, $\Delta_f \neq \overline{\Delta_p}$. In the next section, we enumerate all nine possible relationships between Δ_f and $\overline{\Delta_p}$ and analyze how often each occurs in practice.

III. Relating Δ_f and $\overline{\Delta_p}$

There are nine possible relationships between $\overline{\Delta_p}$, Δ_f, and Δ (Figure 3). Each relationship provides unique information that may help identify the regression failure cause. Before enumerating these relationships, we first discuss two code samples to provide an intuition for the sets of changes we study.

A. Illustrative Examples

Figure 2 shows code examples that illustrate two of the counterintuitive relations between Δ_f and $\overline{\Delta_p}$. The example in Figure 2(a–d) corresponds to case ③ from Figure 3, in which Δ_f is larger than $\overline{\Delta_p}$. The p_1 code version in Figure 2(a) passes the test in Figure 2(i), but the two changes highlighted in Figure 2(b) — an incorrect update to the `getPi` method, and a new call to `getPi` — cause the test to fail. Here, as shown

in Figure 2(c), $\Delta_f = \Delta$ since just breaking `getPi` will not break the test without also adding a call to `getPi`. However, $\overline{\Delta_p}$, highlighted in Figure 2(d), consists of just one of the changes in Δ_f— the call to `getPi`. The failure cause lies in $\Delta_f \setminus \overline{\Delta_p} = \Delta_p$. This phenomenon is called *interference* [10].

The example in Figure 2 (e)–(h) corresponds to case ④ from Figure 3, in which $\overline{\Delta_p}$ is larger than Δ_f. The p_2 code version in Figure 2(e) passes the test in Figure 2(i), but three changes highlighted in Figure 2(f) — an update to the `toString` method and two changes that cache the result of a circle's area — cause the test to fail. There are two independent failure causes: incorrect area computation and incorrect "Circle" and "Area" capitalization. Here, Δ_f in Figure 2(g) consists of only one change, and contains neither of the failure causes! (The test still fails, but for a different reason.) Meanwhile, $\overline{\Delta_p}$ in Figure 2(h) contains the buggy `toString` modification.

B. Voldemort Study

Figure 3 enumerates all the nine possible relationships between $\overline{\Delta_p}$, Δ_f, and Δ.[3] To understand the practical frequency of these relationships, we analyzed Voldemort [7], an open-source distributed key-value storage system.

Voldemort's development process has two properties that are important to our study: it has an extensive test suite, and it does not enforce strict commit standards — commits are allowed to break tests. We considered three randomly chosen ranges of roughly 100 commits each: commits numbered 3a64322–83668c1, f3cd4f9–7376cc6, and f92c899–4c49cf6. Due to branches within the history structure, these ranges contained 109, 97, and 99 parent-child pairs, respectively.

Step 1. We first ran the test suite on each revision and identified all parent-child pairs with at least one test that passed in the parent and failed in the child. We observed 45 such pairs — each pair represents a regression failure. For each of these pairs, we computed Δ, making sure no changes to the tests occurred (which could have caused the regression failures).

Step 2. We then used a file-level change granularity[4] to exhaustively consider all subsets of Δ, applying each subset to p. We ran the test suite on the resulting versions that compiled. We considered a compiling version *passing*, if at least one of the tests that failed in p passed, and no other test that passed in p failed. Otherwise we considered the version *failing*.

Step 3. For each regression failure, we identified the minimal Δ_f and the maximal Δ_p (minimal $\overline{\Delta_p}$). We then compared the two to determine their relationship. Figure 3 shows the observed frequencies of relationships. In cases with multiple Δ_f or $\overline{\Delta_p}$, we labeled each Δ_f and $\overline{\Delta_p}$ pair as one of the relationships in Figure 3, and scaled their frequencies accordingly.[5]

[3]Certain other relationships are impossible. For example, $\overline{\Delta_p}$ cannot be the empty set, since this would imply $\Delta_p = \Delta$ and that the maximal passing version is c, but c fails t, while the maximal passing version passes t.

[4]We considered all changes made to one file atomic. Finer granularity provides higher accuracy, but our analysis provides a **conservative bound**: if $\Delta_f \neq \overline{\Delta_p}$ at the file level, the property also holds at all finer granularities.

[5]That is, if one parent-child pair had three different, equal-sized, minimal Δ_f-$\overline{\Delta_p}$ pairs, each contributed only $\frac{1}{3}$ of an occurrence to the frequency of its Δ_f-$\overline{\Delta_p}$ relationship.

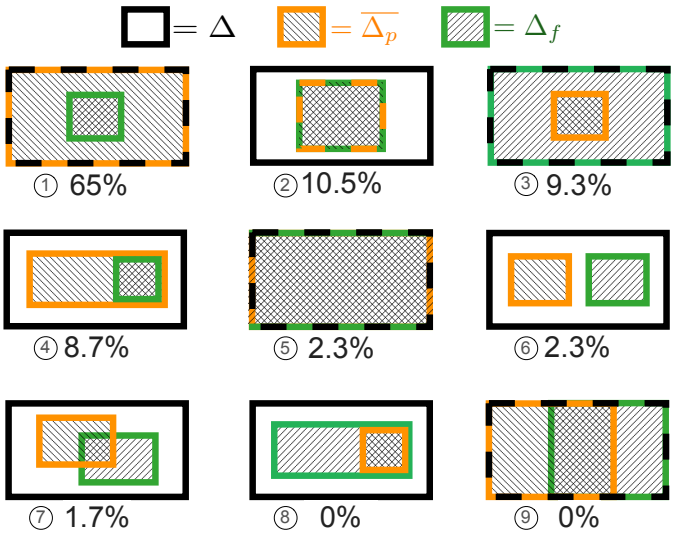

Fig. 3. The nine possible relationships between Δ (black box), $\overline{\Delta_p}$ (orange), and Δ_f (green), annotated with and ordered according to decreasing frequency of occurrence in the Voldemort case study (Section III-B).

C. Study Results and Discussion

Overall, we found that of the 45 regression failures we identified, cases ② and ⑤ in Figure 3 — the cases for which $\Delta_f = \overline{\Delta_p}$ — occur only 12.8% of the time. The other, counterintuitive cases occur 87.2% of the time, indicating that examining $\overline{\Delta_p}$ is at least as important as Δ_f, and provides new information not contained in Δ_f.

Of the nine possible relationships in Figure 3, only seven occurred in our study. In the most common case, ①, which occurred for 65% of the regression failures, all compilable subsets of Δ failed the test. For both this case and case ④, which occurred for 8.7% of the regression failures, $\Delta_f \subset \overline{\Delta_p}$. Thus, reporting $\overline{\Delta_p}$ would include more information than reporting Δ_f, which is what delta debugging reports for these cases.[6] Identifying whether this information would improve developer debugging speed or quality remains as future work, though the fact that the changes in $\overline{\Delta_p}$ are not in the maximal set of changes that keeps the test passing suggests that they are relevant in debugging. For example, in the bug in Figure 2 (e)–(h), $\overline{\Delta_p}$ contains a failure cause, whereas Δ_f does not.

The intuitive case, ②, with $\Delta_f = \overline{\Delta_p}$, occurs for only 10.5% of the regression failures. This finding supports our hypothesis that interactions among changes are complex enough that Δ_f is rarely identical to $\overline{\Delta_p}$, and that both are worth considering when debugging. When $\Delta_f = \overline{\Delta_p}$ (as in this case), this information suggests that the failure cause is in those sets.

For 9.3% of the regression failures (case ③), Δ_f fails to localize the failure cause ($\Delta_f = \Delta$), whereas $\overline{\Delta_p}$ does localize it ($\overline{\Delta_p} \subset \Delta$). Because $\overline{\Delta_p}$ is compilable, here, applying the changes in $\overline{\Delta_p}$ *must* pass the test (otherwise, Δ_f would equal the smaller $\overline{\Delta_p}$). Therefore, Δ_p and $\overline{\Delta_p}$ split Δ_f into two disjoint sets, both of which pass the test. The buggy behavior occurs

[6]If $\overline{\Delta_p}$ fails the test, delta debugging may handle this as a special case and report something larger than Δ_f.

because of the interaction between the two sets of changes, and not in either of the sets on its own. Delta debugging calls this property of changes *interference*. While there is value in considering both Δ_f and $\overline{\Delta_p}$ during debugging, in this case, it can also be helpful to know that Δ_f can be split into two interfering subsets.

For 4.0% of the regression failures (2.3% in case ⑥ and 1.7% in case ⑦), Δ_f and $\overline{\Delta_p}$ disagree on which changes are responsible for the failure. These cases are counterintuitive and particularly interesting. In case ⑥, the minimal failing version of the code is a subset of the maximal passing version: $\Delta_f \subset \Delta_p$. This means there are changes in Δ_p that are not in Δ_f and these changes suppress the failure in Δ_f. This violates delta debugging's requirement that adding changes to a set of failing changes cannot make the test pass. As a result, a single execution of delta debugging will not find this Δ_f, though iterative runs over the right subsets would. In case ⑦, some changes are in both $\overline{\Delta_p}$ and Δ_f: $\Delta_f \cap \overline{\Delta_p} \neq \emptyset$. Considering $\Delta_f \cap \overline{\Delta_p}$ before the larger Δ_f may lead to improved bug localization, though the changes in Δ_f and $\overline{\Delta_p}$ can both aid the process.

Finally, for 2.3% of the regression failures (case ⑤), neither Δ_f nor $\overline{\Delta_p}$ localize the bug. There exist no subset of Δ that compiles (if one existed, it would either pass or fail the test and reduce either $\overline{\Delta_p}$ or Δ_f, respectively).

Cases ⑧ and ⑨ did not appear in our case study. For ⑧, $\overline{\Delta_p} \subset \Delta_f$, so $\overline{\Delta_p}$ provides no new information that is not in Δ_f. It may, however, help identify which changes to examine first. Similarly, in case ⑨, $\Delta_f \cap \overline{\Delta_p}$ may be worth examining first, which may be helpful because each change in Δ is either in Δ_f or in $\overline{\Delta_p}$.

IV. Related Work

When regression failures occur, developers may be interested in knowing which recent changes they should examine. Prior work considered a variety of options, including changes related to the most cross-cutting concerns [1], changes to modules with the highest churn [3], changes made to recent changes [13], changes to modules with the most dependencies [12], and the smallest subset of the recent changes that exhibits the test failure [10].

The work most closely related to ours is delta debugging [10], which computes (or in various efficient ways approximates) Δ_f. Ren et al. [5] introduce an alternate but similar approach to determine Δ_f, and Zhang et al. [11] combine those two approaches to improve efficiency. Our work focuses on whether considering a closely related set, $\overline{\Delta_p}$, and its relationship with Δ_f, can improve bug localization. Further, many of the efficiency techniques from delta debugging can make our approach more efficient, as we discussed in Section III-C.

While most change impact analysis work has also focused on Δ_f [2], some has considered Δ_p, although not for bug localization. Wloka et al. [8] have used change impact analysis to find changes that are considered safe to commit to version control repositories or to share with other developers. This work begins to explore the value of Δ_p, though our focus is quite different. We believe that analyzing the relationships between Δ_f and $\overline{\Delta_p}$ can aid debugging and have demonstrated that in practice, $\overline{\Delta_p}$ and Δ_f often contain different information about the regression failure cause.

Tools and techniques that build on results of finding Δ_f, including automated fault localization techniques [4], [5], [6], are complementary to our work and can likely be improved by considering the relationship between $\overline{\Delta_p}$ and Δ_f.

V. Contributions and Emerging Future Work

This paper considers the relationship between two subsets of changes relevant to diagnosing regression failures: Δ_f, the minimal set of changes that produces the failure, and Δ_p, the maximal set of changes that does not produce the failure. Counterintuitively, these sets are often not complementary; complex dependencies force some changes to be in neither set and some to be in both. In evaluating 45 real-world regression failures from an open-source project's history, we found that in 87% of the failures, the two sets were non-complementary, and in 78% of the failures, the complement of Δ_p contained changes relevant to debugging that were not in Δ_f.

Our preliminary results support our hypothesis that Δ_f and Δ_p relate in complex ways and that both are relevant to debugging. Future work will we check if our findings generalize to a broader set of regression failures, and will examine the relationships between Δ_f and Δ_p at finer change granularity.

Our finding can improve several existing bug localization techniques that have previously focused only on Δ_f. Further, considering the complex relationships between Δ_f and $\overline{\Delta_p}$ may lead to new bug localization techniques. Ultimately, we hope to empirically verify that these bug localization techniques improve debugging speed and quality.

References

[1] M. Eaddy, T. Zimmermann, K. D. Sherwood, V. Garg, G. C. Murphy, N. Nagappan, and A. V. Aho. Do Crosscutting Concerns Cause Defects? *IEEE TSE*, 34(4):497–515, 2008.

[2] B. Li, X. Sun, H. Leung, and S. Zhang. A Survey of Code-Based Change Impact Analysis Techniques. *Software Testing, Verification and Reliability*, 2012.

[3] N. Nagappan and T. Ball. Use of Relative Code Churn Measures to Predict System Defect Density. In *ICSE*, pages 284–292, 2005.

[4] X. Ren, O. C. Chesley, and B. G. Ryder. Identifying Failure Causes in Java Programs: An Application of Change Impact Analysis. *IEEE TSE*, 32(9):718–732, 2006.

[5] X. Ren, F. Shah, F. Tip, B. G. Ryder, and O. Chesley. Chianti: A Tool for Change Impact Analysis of Java Programs. In *OOPSLA*, pages 432–448, 2004.

[6] M. Stoerzer, B. G. Ryder, X. Ren, and F. Tip. Finding Failure-Inducing Changes in Java Programs using Change Classification. In *FSE*, pages 57–68, 2006.

[7] Voldemort. http://project-voldemort.com, 2012.

[8] J. Wloka, B. G. Ryder, F. Tip, and X. Ren. Safe-Commit Analysis to Facilitate Team Software Development. In *ICSE*, pages 507–517, 2009.

[9] A. Zeller. *Why Programs Fail: A Guide to Systematic Debugging*. Morgan Kaufmann, 2nd edition, 2009.

[10] A. Zeller and R. Hildebrandt. Simplifying and Isolating Failure-Inducing Input. *IEEE TSE*, 28(2):183–200, 2002.

[11] S. Zhang, Y. Lin, Z. Gu, and J. Zhao. Effective Identification of Failure-Inducing Changes: A Hybrid Approach. In *PASTE*, pages 77–83, 2008.

[12] T. Zimmermann and N. Nagappan. Predicting Defects with Program Dependencies. In *ESEM*, 2009.

[13] T. Zimmermann, P. Weißgerber, S. Diehl, and A. Zeller. Mining Version Histories to Guide Software Changes. *IEEE TSE*, 31(6):429–445, 2005.