Change Fault Localization Using Test Execution Profiles

Jay Liu
Department of Computer Sciences
The University of Texas at Austin
Austin, TX, USA
jayliu@cs.utexas.edu

ABSTRACT

Regression tests ensure that software systems retain correctness in their existing functionalities while the systems evolve. During development, regression tests allow the programmer to quickly identify problems. If there are any new failed tests, since those failures surfaced after new changes to the code base, we need to identify where this failure originated in our code changes. In this paper, we combine regression test results with changes in both the source code and test execution profiles to narrow down and identify the changes that may have caused test failures.

Categories and Subject Descriptors

D.2.5 [**Software Engineering**]: Testing and Debugging—*Debugging aids*

General Terms

Fault localization

1. INTRODUCTION

Software systems frequently undergo large changes to its source code. These changes result from some requirements that either address bugs in the existing systems, or create additional new features. To ensure correctness of these changes, it is important to have tests that provide quantitative feedback. During the software testing phase, not only is it important to inspect whether the new requirements have been met, it is just as important to ensure that the existing functionalities have not been adversely impacted. These regression tests are important in making sure that known use cases are not impacted, and previously fixed bugs are not introduced again.

Testing software can be a challenging and time consuming process. In the event of a test failure, it can be difficult to find its root causes, especially in larger systems where the execution goes through large number of steps. This issue is exacerbated in the case where the developer who works on a requirement is typically only interested in having the requirement met, and either chooses not to have or simply does not have exposure to parts of the system that are beyond those that she is interested in changing. In the case where regression tests fail, it can be difficult to understand how the new changes adversely affected existing functionality. A developer, when encountering a test failure while contributing code, is interested in the question: "What have I changed that has caused this failure?" We attempt to answer this question using a combination of a testing framework, an execution profiler and various source code version differencing tools to find and highlight the changes that most likely contributed to the test failure.

There are ways, even for those who are unfamiliar with the failed tests and the tested components, to troubleshoot the problem. The most obvious is that if some particular tests pass for the existing code base, but are now failing after the changes have been added, it can be deduced that all or some subset of the new changes led to the failure of these tests. Note that this does not necessarily mean that the new changes are the root cause of the failures; they may be using an existing functionality that was already buggy but not yet uncovered. Since the developer will be familiar with his own changes, he will be able to inspect the code and reach a conclusion as to what is causing the bug.

In addition to the visible code level changes, the execution profile while running the test may itself provide clues. With the same test input data, a test for a deterministic program will take distinct execution paths that lead either to pass or failure. Therefore, changes in the execution flow will provide additional information that may help with debugging the program

In this paper, we will investigate the feasibility of using source code level changes and test execution profile changes to identify possible fault-inducing changes. We do so by first surveying existing literature on fault localization, especially those that emphasize regression tests and their execution profiles. We create a software tool based on existing well-known testing tools that implement our algorithm for Java programs. We base our algorithm not only on textual code differences, but also attempt to take advantage of logical differences to provide more readable output.

Beyond simple feasibility, we would also verify that the identified changes achieve good accuracy. The result set should contain as few non-fault-inducing changes as possible, while still maintaining safety. A result set is *safe* if it is a superset of all problem changes [12]. In other words, all possible fault-inducing changes are contained in the result set, even if it is necessary to include false positives.

2. BACKGROUND AND RELATED WORK

2.1 Regression Test Profile

It is now common to find most software projects utilize a large number of tools to not only simplify development, but also increase quality. To automate quality assurance, well-written tests are an important part of any project. These tests evolve like the software application itself, and their executions have become an important part of the build process. There are two important metrics testing can provide. The first is obviously whether the code is executing as expected by having the tests either pass or fail. This binary indicator can quickly point to problems. However, it does not provide further insight into the problem itself. A second metric that has become widespread is code coverage. As the tests are run, an execution profile is created to record which lines in the code have been in fact run by the tests, and how often. This information is often used to decide whether sufficient number of appropriate tests have been written.

In practice, test results and test execution profiles have largely been used independently. However, by correlating test failures with their execution profiles, it is possible to provide important insight into the exact causes of these failures. The underlying idea is that a test that fails necessarily has a different execution path from one that passes, barring any non-deterministic behaviors. Based on this, we rely on fault-localization techniques based on targeting specific execution *slice* and *dice* [1].

An instance of a test case has an execution profile called “slice”. An instance that passes and an instance that fails given the same input have different slices that do not entirely overlap with each other. The parts that do not overlap are called “dices”. The bug can be in either the overlapping region of the slices, or in the dice of the failed instance. It was found that fault-localization can be effective even when it is restricted to the dice of the failed instance. In other words, failures can be identified by inspecting deviations in execution profile of failed instances from instances that pass, rather than inspecting the entire execution.

Fault localization attempts to identify code that is likely causing the tests to fail. To quantitatively measure this likelihood, a number of algorithms have been proposed. Some of them are presented and compared in [7]. One example, Tarantula [8], directly relates the statements that a test instance runs with the result of that instance. Each time a test instance is run, a code statement can be in three states: it was part of a pass instance, a failure instance, or it was not executed at all for that instance. Extrapolating from this data, we can calculate the likelihood of this line of code running when an instance passes or fails. A statement that is likely to run when a test fails is likely buggy. Using heuristics, we can set a threshold above which we assert that the code is faulty and perform inspection.

Tarantula is very suitable for test driven development, where we run the tests for the first time against an implementation, and identify problem areas. In this paper, we look at regression tests, which afford us access to code changes. The difference is that regression tests have typically previously been run successfully. Therefore, rather than having to look at the entire source code, we can narrow down to the changes in the source code, and observe which ones caused the tests to start failing. Having a smaller quantity of code to look at allows us to provide more concise hints to the developer.

The idea of using code coverage can be extended to include branching behaviors [13].A similar work in [10] also emphasized the role of the execution profile of passed tests.

2.2 Source Code Differencing

Using source code differencing data between current version of the code and its previous version can be an effective debugging tool. Disciplined use of source version control allows us to assert that the cause of failure must reside in the changes between the two versions. Automatic “delta debugging” [15] is a technique where we systematically roll back changes until we find the smallest subset of changes that will still cause the tests to fails (“minimal failure-inducing change set”). Breaking up this set will cause the tests to pass. If the change is large, where searching for such a set can be expensive, it may still be a challenge to isolate the problematic code. By using test execution profile, we hope to simplify the process by narrowing down the change set to those changes that appear interesting.

2.3 Logical Differencing

An alternative approach to using textual differences is to use logical differences. ChangeDistiller [5] is an Eclipse plug-in toolkit that accepts two versions of a source file, generates abstract syntax trees, and produces a summary of the logical differences between the two trees. Such logical breakdown allows for greater visibility and granularity into the changes. While a line-level difference may only see that the declaration of the method has changed, logical differencing can identify fine-grained changes such as changes of the modifier, its derivability, and its argument types, etc. It will also be able to discover moved entities in a way that textual differencing may miss. As a result, logical differencing can lead to fewer false positives.

As applied to fault-localization, Chianti [14] uses call graphs to identify code changes that affected the outcome of the tests. It detects three particular types of changes seen by tests: changed method, deleted method, or changes in lookup calls. While this isolates the particular method that may be in error, a method may still be large and complex enough that it is useful to further investigate at the statement level.

3. APPROACH

We need two versions of the same software program, say versions A and B, which differ by some changes in their source code. A is an older version, and B the new version. We also have a set of regression tests T that can be run against the two versions of the software. We denote the percentage of passed tests over all executed tests on versions A and B to be R_A and R_B respectively. We require our tests to be deterministic, such that R_A is the same given the same

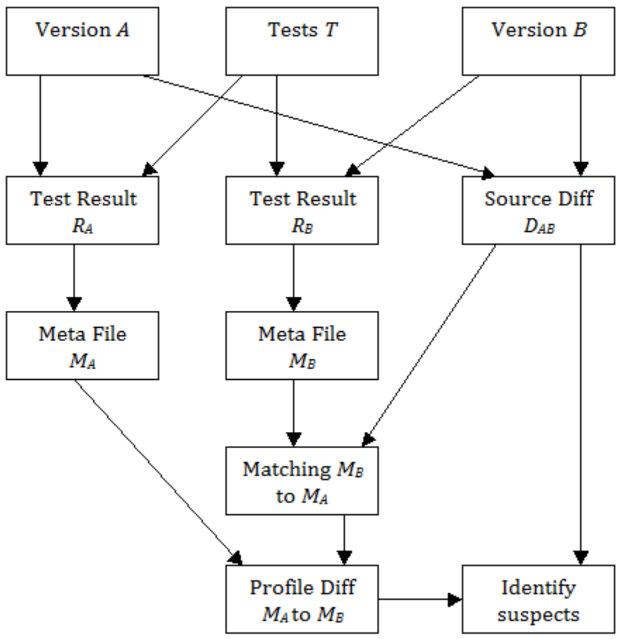


Figure 1: High-level view of solution approach

T and A. Similarly, R_B is the always the same given T and B. Figure 1 gives a visual overview of the approach. The notations and algorithms are explained in detail below.

3.1 Test Execution Profile

The first data set we need to gather is the contribution of each line of code to the test results. Each line in the source code can contribute differently. Some lines do not contribute to test results at all. Ideally all such lines fall into the non-code category, such as comments or blank spaces. However, some lines can be actual program code but still do not contribute to test results because none of the tests execute those lines. The profiling software we use is able to differentiate between non-code and actual untested code. Improving test coverage to decrease the amount of untested code will improve the accuracy of our approach.

We measure the contribution of a line of code by observing whether tests tend to pass or fail when their execution paths include this particular line. To do so, we need to know which tests executed this line of code, and the results of those tests. We give each test an identifier. Using an execution profiler, when a line of code is run by a test, we tag that line with the identifier of the running test. A line can be run by multiple tests, and each test may run a line multiple times. For our purpose, we are only interested in observing the differences when multiple tests execute this line, since running a line multiple times by the same test will all have the same test outcome. Treating each execution by the same test as a separate instance would incorrectly give more weight to tests that run a line of code repeatedly versus tests that only run it once.[1] Therefore all executions due to the same test will be treated as one data point.

[1]This is of course assuming that all tests are equal in importance in a test suite.

With the execution profiler enabled, the entire test suite is executed against the first version, A, of the software. We obtain a set of metafiles M_A, one file for each source code file. For each line in the source code, the metafile records how many tests that executed that line of code passed and how many of those tests failed.

Similarly, we run the same test suite against the second version, B, of the software and collect the metafiles M_B. Tests that pass in the first version of the software could fail in the second version. Our algorithm tries to identify fault-inducing changes by observing which line-level changes have made the test results worse than before.

If the set of tests have identical test results when run against both versions of the software, i.e. $R_A = R_B$, or if tests that have failed in the earlier version are now passing in the new version – in other words our changes are in fact improving the test results, $R_A < R_B$ – then we may not be able to observe any erroneous differences. Of course, there may exist incorrect changes if none of the tests execute the changed lines of code, or the number of newly passing tests exceeds the number of newly failing tests. Therefore, our approach works best if the tests are highly granular, and all tests pass when running against version A.

3.2 Differencing

The next step is to perform differencing on the collected data. The two versions of the metafiles cannot be compared directly because the lines in them are different due to source code changes. To compare metafiles, therefore, differencing of the source code needs to be performed first.

Using the *diff* [6] application, we obtain source code line level differences D_{AB}. Such differences, in the form $L_A \chi L_B$, shows how lines L_A have changed to L_B from version A to B. We use the notation l to denote a single line of code, and L to denote a block of one or more lines of code. χ denotes the type of change between the two blocks of codes from the two different versions.

Using this information, we can now line up metafiles M_A and M_B using D_{AB}.In our approach, we are only interested in observing the execution profiles of lines of code that have undergone changes. The execution profiles of a line may be different in version B even if that line had no change. However, since that line was not changed, it could not be the cause of the difference in its execution profile. That execution difference must be caused by somewhere else by a line change. Therefore, it is sufficient to only compare the execution profiles of the changed lines.

For all line changes $L_A \chi L_B$, we find execution profiles for lines L_A and the corresponding lines L_B and observe any changes. If there are changes, then, depending on the change type χ, we may have encountered a suspicious line change. There are three types of χ changes that we can observe in our code. "a" denotes "add". In this case, lines L_B were added after L_A. "c" denotes "change", where lines L_A were replaced by L_B. And "d" denotes delete, where L_A were removed. For example, (12,14c15) means that lines 12 through 14 in A have been replaced by line 15 in B.

We use *suspiciousness* to indicate how likely we believe a line of code could be the cause of the new test case failure. Our goal is to produce a list of *suspects*, which are changes whose suspiciousness score lead us to believe they are the cause of the new test failures. Depending on the actual change type χ, we identify suspects slightly differently.

3.3 Suspect Identification

To identify whether an execution profile difference indicates potentially buggy code, we rely on the Tarantula score system. The Tarantula score, used to measure the suspiciousness σ of the entity e, is expressed in [7] as:

$$\sigma(e) = \frac{\frac{\text{failed}(e)}{\text{total failed}}}{\frac{\text{passed}(e)}{\text{total passed}} + \frac{\text{failed}(e)}{\text{total failed}}}$$

where e is a coverage entity, which in our case is a line of code. failed(e) is the number of tests that executed e and failed, whereas similarly passed(e) is the number of passed tests.

If the score is 0, then this line e was not part of any test that failed, and is therefore not suspicious. If the score is 1, then this statement was not part of any tests that passed, and is therefore very suspicious. The score of 0.5 is ambiguous: this line is as likely to experience failure as the overall test set. To satisfy safety, we consider any score greater or equal 0.5 to be suspicious.

For the possible values of χ, we identify suspects as follows:

1. *Change:* This is the most intuitive case. Lines L_A in A were replaced by lines L_B in version B. The execution profile of L_B of the new version is directly comparable to the profile of L_A in prior version. A change may replace one or more lines with one or more different lines. Collectively, the new changes in L_B should not result in more failures than the code L_A they replace. Therefore, a line $l_B \in L_B$ is suspect if:

   $$\sigma(l_B) > \max_{\forall l_A \in L_A} (\sigma(l_A))$$

   In other words, a line l_B has a higher suspiciousness score than the worst line in L_A. We do not consider new code that fail less often than old code to be suspect even if they still may contain fault. They are not suspects because the changes are causing fewer, not more, failures, and are therefore correct.

2. *Add:* These lines are newly added to version B and are not replacing any lines previously present in A. The execution profile of the new version does not have equivalence in prior version. The suspiciousness of this code is its Tarantula score.

   We can explore an alternative method by observing whether a parent entity of L_B exists in L_A. For example, if $l_B \in L_B$ is added to some function F or to some class C, spanning lines L_P, and L_P exists in A, we can try to identify suspects l_B if:

   $$\sigma(l_B) > \max_{\forall l_P \in L_P} (\sigma(l_P))$$

   Therefore, l_B is only suspicious if it causes more failure than the worst statement in its parent entity. This alternative method, however, risks false negatives because a change can be wrong and introducing additional faults, but it is introducing fewer faults than another unrelated statement in the parent entity. It would therefore violate safety.

3. *Delete:* When a block of code had been deleted in B, there is no new test execution profile to produce and compare. This creates a gap in the metafile comparisons, causing either remaining changes or existing code to be incorrectly blamed. This "omission" creates a problem.

   One way to resolve this is to convert all *Delete* operations into a *Change* operation, where the new changes are in effect no-ops but nevertheless will be able to accumulate execution metrics. Typically, if there is a line preceding the L_A block that is deleted, we make a change to that line by adding a comment. Otherwise, we add a dummy but correct line of code such as `int _B = 0;`.

   We show in our evaluation that this solution does work. Unfortunately, it is not straightforward if the deleted code forms a section where a no-op replacement would be syntactically incorrect. For example, if we remove a method, putting a no-op in its place would not help. This would require a more creative interpretation. For example, if a method were removed, then either it was not called by anyone else before in A, in which case its removal cannot induce fault, or it was called somewhere by L_A, but since it was removed in B, L_A had to be changed or removed, in which case we have a more manageable line change.

3.4 Logical Differencing

In order to make our result more readable, we additionally translate the result set of suspected line changes into logical differences. Whereas a line-level difference will show which line has changed, it may be coarse-grained because a line may contain multiple changes, especially if a line is a concatenation of multiple statements. Logical difference will provide additional insights by breaking the changes down into granular levels, such as `PARAMETER_INSERT` and `METHOD_RENAMING`. It is also able to group multiple line-level changes into one larger logical change: an if-statement block that spans multiple lines can be expressed as `CONDITION_EXPRESSION_CHANGE`.

If no helpful logical differences are present, we will simply show either `STATEMENT_INSERT`, which directly maps to a line-level add, and `STATEMENT_UPDATE`, which corresponds to a line-level change.

3.5 Implementation Details

We rely on two popular development tools for gathering our test execution profiles. JUnit [9] is a unit testing framework for Java. It is used to execute all tests. JUnit test cases have three possible outcomes: *pass*, *fail*, and *error* when a unexpected execution state was reached. For our evaluation, we consider both *fail* and *error* as failures. The JUnit framework was modified so that we can identify each test case. We use the execution time of the test case as its unique identifier. This is valid since our tests are executed sequentially. The result of each test case, either pass or fail, is also recorded.

We run the tests with an execution profiler, for which we use the code coverage tool Cobertura [2]. The execution profiler allows us to perform an action when a particular line in the source code had been executed. To associate the execution with a test case, we record the timestamp of when it was executed. By observing the timestamp identifiers of the tests, and the timestamp of when a line of code was executed, we can crossreference them to determine which tests executed a particular line of code. Then by looking up whether those tests passed or failed, the metafile entry is made for that line.

Finally, to produce logical differences, we use ChangeDistiller, which takes two versions of a file and produces a list of logical differences. We map our line-level changes to the logical differences at those lines to produce logical difference result for the suspected changes.

4. EVALUATION

We now describe our evaluation of the algorithm presented above.

4.1 Seeding Faults

For our tests, we compare two versions of the same software program, where one version contains more faults than the other. To conduct our experiments, we obtain a working copy of the software program, which we use as version A. Then, we make a copy, and introduce arbitrary code changes. Some changes are benign and do not affect the program behavior. Such examples include correct refactorings such as renaming of variables, or extracting methods from code blocks. These actions, while they produce differences in code and different paths in the execution profiles, do not change the test results.

We can also introduce changes that intentionally break the program, causing tests to fail. Such changes are called "seeded faults" [4]. Seeded faults are bugs artificially inserted into the code to modify the execution behavior. We can control the types and the frequency of the faults that we insert into a correct program, so that we can inspect various aspects of our fault-localization algorithm based on predictable input. However, fault seeding does represent a threat to external validity [4].

In contrast, without seeding faults, we can review the history of the software that we work with to find naturally occurring faults. Those are faults that were unintentionally committed by developers and checked in before they were caught. While such naturally occurring faults may be more representative of the kind of faults that we should expect in a software system, identifying such faults can be expensive since in a good development process, a developer ensures that all faults are eliminated before the code is checked in, leading to a dearth of available naturally occurring faults.

4.2 Measurement Criteria

In our approach, we attempt to find fault-inducing changes. A "change" in this case refers to a block of lines L_B. We assign:

$$\sigma(L_B) = \max_{\forall l_B \in L_B} (\sigma(l_B))$$

Therefore, if any of the lines in L_B is considered suspicious, the change block L_B is considered suspicious. Part of the reason for identifying blocks of lines instead of individual lines themselves is to reduce ambiguity. For example, introducing a new if-statement block may involve multiple lines, and many of the lines may report to be suspicious. It can be ambiguous to pick which specific lines are causing all the failures because all lines in a change block are new and are interacting with each other to produce faults. We are assuming that lines in such a block are related to the same feature, which for incremental code changes is most likely true.

We summarize our experiment results in Table 1. "Oracle" is the number of fault-inducing changes for which execution profile is available. This is the number of faults that we expect our algorithm to identify. "Oracle" does not include faulty changes that are not covered by the execution of any test, since any such change cannot be expected to be found by the algorithm. "Suspicious" is the actual number of changes identified as suspects by our algorithm. "Match" means the number of "suspicious" changes that correspond to changes identified by "oracle". When "safe" is "Yes", it means that "oracle" is a subset of "identified". Since the untested changes are not part of "oracle", safety is only considered for all discoverable changes.

To quantitatively measure the correctness of the algorithm, we use precision and recall.

Precision measures whether the identified suspects are indeed fault-inducing, i.e. part of "oracle". It can be expressed as:

$$F_P = \frac{\#\text{ correctly identified suspects}}{\#\text{ all identified suspects}}$$

Recall measures whether all problematic changes of code in "oracle" were identified as suspects. It can be expressed as:

$$F_R = \frac{\#\text{ correctly identified suspects}}{\#\text{ all fault-inducing changes}}$$

Effectiveness measures how many changes we were able to eliminate through our technique. Since the goal is to narrow down the number of changes to inspect, we would like to reduce as many of them as possible. The more changes we are able to eliminate by marking them as not suspicious, the more we are able to focus on the remaining suspicious lines. Effectiveness is expressed as:

$$F_E = \frac{\#\text{ changes} - \#\text{ all identified suspects}}{\#\text{ changes} - \#\text{ fault-inducing changes}}$$

where due to safety constraint of:

$$\#\text{ all identified suspects} \geq \#\text{ fault-inducing changes}$$

we always have $F_E \leq 1.0$, or it is otherwise reported as not applicable.

Set	Run	R_A	R_B	Changes	Oracle	Suspicious	Match	F_P	F_R	F_E	Safe?
1	1	1.0	0.67	4	1	1	1	1.0	1.0	1.0	Y
2	1	0.83	0.33	26	1	2	1	0.5	1.0	0.96	Y
2	2	0.98	0.83	22	2	4	2	0.5	1.0	0.9	Y
2	3	0.98	0.66	6	2	4	2	0.5	1.0	0.5	Y
2	4	0.98	0.63	12	7	6	5	0.83	0.71	N/A	N
2	4a	0.98	0.98	6	2	1	1	1.0	0.5	N/A	N

Table 1: Experimental Results

4.3 Set 1

To illustrate our algorithm, we created a simple correct program, as shown in Figure 2.[2] A set of 3 tests was created against this program, and all tests passed.

The program was then modified. Some changes do not alter the outcomes of the tests, such as a rewrite of the `sum` function, but one change where `absolute` function was changed incorrectly did cause a test to fail.

The textual differencing result is { (42c42,43), (52a54,56), (54c58), (56c60) }. It can be observed that the number of changes is 4, spanning 6 lines.

During testing, the first version of the program passed all tests. In the second version, lines 40 and 43 participated in the execution of a test failure, and line 47 was no longer called. The test execution profiles for the two programs show the following differences: { (40c40), (42c42,43), (46c47), (53a55,57) }.

Our algorithm correctly identified line 43 as the culprit with a score of 1.0, because it is in both textual differences, and it participated in a failed test. Line 40, on the other hand, was not suspected, even though it was also part of the failed test. This is because line 40 was not changed according to the source code difference. The suspects we identified is exactly the "oracle" set. Therefore, $F_P = 1$ and $F_R = 1$. We narrowed down from 4 changes to 1 change, so $F_E = \frac{4-1}{4-1} = 1.0$

This example, despite its simplicity, shows that the identified approach can be applied to find suspicious fault-inducing changes. Where the change set is small, it is highly probable that the change set itself is already the minimal failure-inducing change set referred to earlier. Identification of the suspicious code should be guaranteed.

To provide higher level visibility into the nature of the change, we crossreference line 43 with the result of the ChangeDistiller logical differencing result. Line 43 was part of a `STATEMENT_UPDATE`.

4.4 Set 2

To evaluate our algorithm further, we employ a larger project, in this case Mozilla Rhino [11], which is an open-source JavaScript engine written in Java. It consists of 190 Java classes, totalling 80,000 lines of code. Rhino release has a total of 155 test cases. Some of its regression tests and the bug fixes for which they were intended were documented by [3]. Many of the bug fixes were very small changes, and as we have shown earlier, such cases are trivial to solve using our algorithm. Also, due to aggressive release cycles, many of bug fixes are no longer present in their archived form in the latest version of Rhino. Therefore, it is necessary to create custom copies of the software with each having changes randomly inserted.

4.4.1 Run 1

In the first evaluation, 6 of the 155 available tests were selected. Running against a freshly retrieved copy of Rhino, we passed 5 of the tests and failed 1. $R_A = \frac{5}{6} = 0.83$ is the baseline.

Next, various modifications were introduced to three files that are at the heart of the system. In all, 26 changes spanning 27 lines were introduced. These changes included renaming of two methods, which are benign but affected a number of statements. In addition, two logical errors were introduced: one that causes errors in the execution path, another that none of the tests checks. Realistically, then, only the first bug is catchable, so the "oracle" has size 1. Running the regression tests again yielded 2 passes and 4 failures.

The algorithm eventually identified 2 changes that are suspicious, one with a Tarantula score of 1.0 and another of 0.5. The former correctly identified the location of the logical error whose discovery is expected. The latter is a method call to one of the renamed methods, which is not a bug. In this case, $F_P = \frac{1}{2} = 0.5$ and $F_R = 1$.[3] By narrowing down to 2 changes from 26 changes, $F_E = \frac{26-2}{26-1} = 0.96$.

4.4.2 Run 2

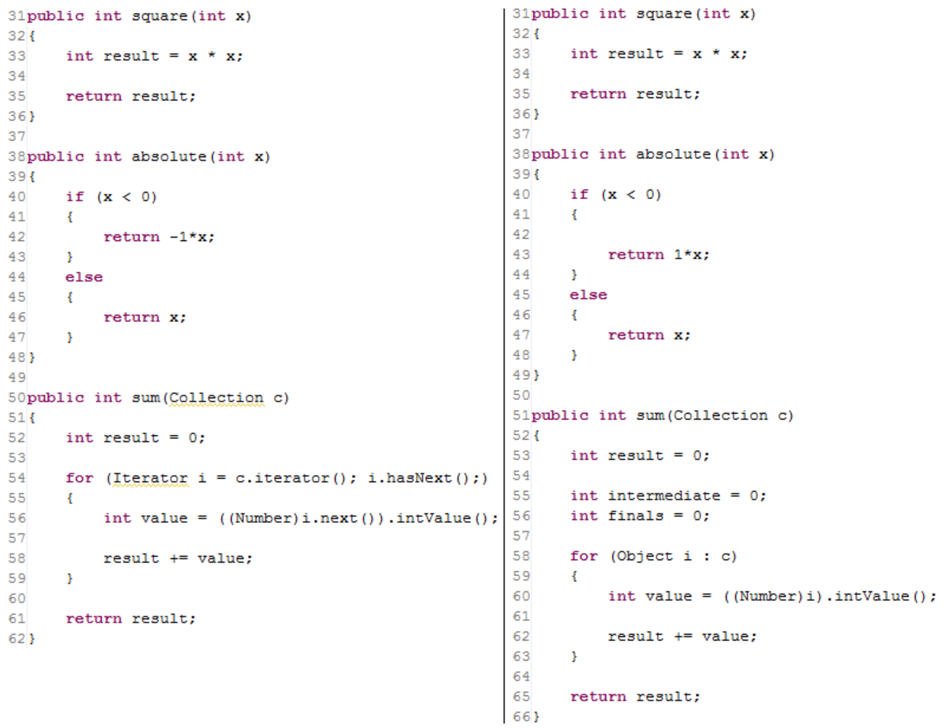
We run all 155 available tests. The baseline result is 3 failures, so $R_A = \frac{152}{155} = 0.98$. We seeded 22 code changes spanning 24 lines.

There were two material changes where assignments were incorrectly changed, and one variable renaming. We also tested a "delete" change: to convert a "delete" to a "change" difference, we modified the line before the deleted statement by adding comments to it, thereby converting a change that would have appeared as {(80d79)} into {(79,80c79)}.

Our approach found 4 suspicious changes with scores 0.85, 0.99, 0.99, 1.0. Two of these are expected, and two of them are false positives due to the variable renaming. $F_P = \frac{2}{4} = 0.5$ and $F_R = \frac{2}{2}$. Even though our precision is merely 0.5,

[2]Original version is a sample program from Cobertura.

[3]Since it was expected to find only 1 error.



```
public int square(int x)
{
    int result = x * x;

    return result;
}

public int absolute(int x)
{
    if (x < 0)
    {
        return -1*x;
    }
    else
    {
        return x;
    }
}

public int sum(Collection c)
{
    int result = 0;

    for (Iterator i = c.iterator(); i.hasNext();)
    {
        int value = ((Number)i.next()).intValue();

        result += value;
    }

    return result;
}
```

```
public int square(int x)
{
    int result = x * x;

    return result;
}

public int absolute(int x)
{
    if (x < 0)
    {

        return 1*x;
    }
    else
    {
        return x;
    }
}

public int sum(Collection c)
{
    int result = 0;

    int intermediate = 0;
    int finals = 0;

    for (Object i : c)
    {
        int value = ((Number)i).intValue();

        result += value;
    }

    return result;
}
```

Figure 2: Evaluation set 1. Left program is a correct program. Right program is incorrect.

we narrowed down 22 changes down to a set of 4 changes that includes all error cases.

4.4.3 *Run 3*

This test focuses on adding additional conditions to switch-statements and if-conditions. There are six seeded changes, two of which were faults. The F_E score was lower in this run than the previous runs because the proportion of faulty changes to total number of changes is high. Therefore, there are fewer changes available for the algorithm to eliminate. A single false positive can significantly lower the F_E score.

4.4.4 *Run 4*

There were a variety of changes, including refactoring, adding new methods, and changing conditional statements. This run differs from the prior runs due to its quantity of faulty changes, which is large both in absolute number and as a proportion to total number of changes. This run did not produce a result that satisfied safety. There were a total of 7 faulty changes out of a total of 12 changes. However, we were only able to detect 5 of the faulty changes. We examine why 2 faults were missed.

To understand why these faults were missed, we have to inspect their execution profiles. Two reasons were identified, both of them due to unexpected interactions of the faulty changes.

In the first case, another fault-inducing change causes a test to abort prematurely. Since the test was verifying intermediate output, it fails as soon as it enters an execution state that is unexpected. The test was not completed, causing lines that were previously covered by the test to be skipped, among which was our missed fault.

The second case was somewhat similar. Another fault inducing change causes our missed faulty change to be executed less often, and also failing less often. This led to a lowering of the Tarantula score as compared to the baseline, causing it to be unsuspected.

This evaluation shows that with a large number of faults, our approach can catch some of the faults, but because some faults completely change the way the program works, previously tested code may no longer be tested in a faulty execution. Therefore, it is necessary to apply fault-localization repetitively. When faults are identified, they should be fixed, and then the fault-localization algorithm re-run to find if there are other lingering faults.

After reverting the correctly identified faults, run 4a was able to identify 1 of the 2 missing faults in run 4.

The last fault, incorrectly setting a value to null, remained elusive. It had the same execution profile as the correct version, and therefore indistinguishable. It is possible that either the tests were not verifying the behavior correctly, suggesting improvements to the tests were needed, or that the change, while logically incorrect, was not causing any faults in the actual program (maybe the variable it sets is not used), and arguably not fault-inducing and should be excluded from “oracle”.

4.5 Logical Differences

Finally, we make use of ChangeDistiller to identify the logical changes that are associated with the line changes. The

Line-level Difference	ChangeDistiller Output	Output after Mapping

49c49
< InterpreterData idata;
—
> InterpreterData intdata;
	1909:22 ATTRIBUTE_RENAMING	Line 49 is due to ATTRIBUTE_RENAMING

Table 2: Example of logical difference descriptions.

output of ChangeDistiller gives logical change descriptions and their associated range of characters in the source file. Using this as input, we find the lines in the source file that correspond to the location of those characters. An example of this processing is shown in Table 2. In the first column, a textual *diff* result tells us that line 49 had been changed. In the second column, ChangeDistiller describes an `ATTRIBUTE_RENAMING` change starting from character 1909 spanning 22 characters. Our mapping found that these 22 characters were all on line 49. Since we were interested in line 49, we print the result in the third column that says on line 49, there is this renaming change. A developer can quickly see what kind of changes occurred on a line before digging into source files and manually inspecting the changes.

The evaluation of this step is somewhat subjective. If the ChangeDistiller output is correct, in our evaluation we found that we are always able to relate our line changes to one of the logical differences. However, sometimes we do find that the ChangeDistiller gives a character range that is different from what can observed in the actual source code, and therefore causes our mapping to fail incorrectly. This appears to be an issue with ChangeDistiller and not with our mapper.

ChangeDistiller is able to supply a logical change description for every change. Therefore, we can map all suspicious lines to at least one description.

4.6 Performance and Scalability

The most significant amount of time is spent on running the regression tests and collecting execution profile data. The various differencing steps take minimal time. Our suspect identification algorithm also runs extremely fast. Since we compare each source file individually, this algorithm can scale linearly with the number of changes and the number of changed files.

4.7 Threats to Validity

One of the biggest threats is external validity due to seeded faults. To allow us to control for the presence and quantity of faults, we chose a production software program and injected faults to its working copy. The results that we obtain by using this technique means that the metrics are really only representative of the specific software programs that we have used, and the specific faults that we seeded. It is always possible that the results here are not typical and are not reproducible on another software program or with different bugs.

Also, we proposed three different numeric measures – precision, recall and effectiveness – to help us gage the effectiveness and correctness of our algorithm. While we believe they are beneficial, they pose a threat to conclusion validity. If the algorithm scores highly on those measurements, we are drawing the conclusion that the output of the algorithm are interesting to developers and indicate that our algorithm is useful. However, output with scores that we believe are theoretically high enough to be useful may still not satisfy users in an actual development environment. It may require real user testing to find out what qualities in the output of a fault-localization system are desirable.

5. CONCLUSION AND FUTURE WORK

The evaluation instances show that the proposed algorithm can be employed to identify potentially fault-inducing changes. In general, all metrics are at or above 0.5. $F_P \geq 0.5$ signifies that our algorithm is able to find fault-inducing changes with a limited number of false positives. $F_R \geq 0.5$ means that we were able to identify most of the faults. When safety is met, we also achieve $F_E \geq 0.5$. While F_E is dependent on the ratio of "oracle" over total number of changes, in general, we are able to eliminate many correct changes so that a programmer does not need to investigate correct code when troubleshooting. Experiments with different programs need to be conducted to more accurately capture the behavior of the algorithm as applied to different programs.

Solution to handle "delete" changes warrants further investigation and evaluation. One approach that may help in that regard is to use logical units as a coverage entity e and find suspects based on $\sigma(e)$ of such logical units. We may use a dynamic graph such as a CFG, and observe how the suspiciousness score changes along the execution path on which the removed entity used to lie.

It certainly should be noted that this solution in general is only as good as the quality and coverage of the regression tests. As our evaluation showed, a bug can still be introduced and go unnoticed if it is at a location where no regression test reaches or checks.

Lastly, we were surprised at some unexpected output from ChangeDistiller, and would like to investigate further why some reported character ranges are different from what is in the source code file. We also would like to perform some user study of whether presenting the logical change description makes this a more useful tool than simply presenting the suspicious line numbers.

6. REFERENCES

[1] H. Agrawal, J. Horgan, S. London, and W. Wong. Fault localization using execution slices and dataflow tests. In *Proceedings of the Sixth International Symposium on Software Reliability Engineering*, pages 143–151, 1995.

[2] Cobertura. *http://cobertura.sourceforge.net/*.

[3] V. Dallmeier and T. Zimmermann. Extraction of bug

localization benchmarks from history. In *ASE '07: Proceedings of the 22nd IEEE/ACM International Conference on Automated software engineering*, pages 433–436, New York, NY, USA, 2007. ACM.

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

[5] B. Fluri, M. Wuersch, M. PInzger, and H. Gall. Change Distilling: Tree Differencing for Fine-Grained Source Code Change Extraction. *IEEE Transactions on Software Engineering*, 33(11):725–743, 2007.

[6] GNU Diffutils. *http://www.gnu.org/software/diffutils/*.

[7] J. A. Jones and M. J. Harrold. Empirical evaluation of the tarantula automatic fault-localization technique. In *ASE '05: Proceedings of the 20th IEEE/ACM International Conference on Automated Software Engineering*, pages 273–282. ACM, 2005.

[8] J. A. Jones, M. J. Harrold, and J. Stasko. Visualization of test information to assist fault localization. In *ICSE '02: Proceedings of the 24th International Conference on Software Engineering*, pages 467–477. ACM, 2002.

[9] JUnit. *http://www.junit.org/*.

[10] C. Liu, X. Yan, L. Fei, J. Han, and S. P. Midkiff. Sober: statistical model-based bug localization. In *ESEC/FSE-13: Proceedings of the 10th European Software Engineering Conf. held jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 286–295. ACM, 2005.

[11] Mozilla Rhino. *http://www.mozilla.org/rhino/*.

[12] G. Rothermel and M. J. Harrold. A safe, efficient regression test selection technique. *ACM Transactions on Software Engineering and Methodology*, 6:173–210, 1997.

[13] R. Santelices, J. A. Jones, Y. Yu, and M. J. Harrold. Lightweight fault-localization using multiple coverage types. In *ICSE '09: Proceedings of the 2009 IEEE 31st International Conference on Software Engineering*, pages 56–66. IEEE Computer Society, 2009.

[14] M. Stoerzer, B. G. Ryder, X. Ren, and F. Tip. Finding failure-inducing changes in Java programs using change classification. In *SIGSOFT '06/FSE-14: Proceedings of the 14th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 57–68. ACM, 2006.

[15] A. Zeller. Yesterday, my program worked. Today, it does not. Why? *SIGSOFT Software Engineering Notes*, 24(6):253–267, 1999.

APPENDIX

A. IMPLEMENTATION USAGE INSTRUCTIONS

The implementation package contains a step by step instruction manual to show how to apply this fault-localization algorithm to any program. The program should be written in Java, and use Ant build system, with JUnit test cases. In addition to modified versions of the JUnit and Cobertura packages, three custom programs are also included:

1. *diffSplitter.pl* separates output produced by *diff* into individual files, one for each class file.
2. *ExecutionProfileDiff.class* is a Java program that calculates Tarantula scores and identifies suspicious lines in the code.
3. *distillerLineMapper.pl* takes ChangeDistiller output and produces logical difference descriptions given a list of suspicious line numbers.