A Survey on Software Fault Localization Techniques

T. Prabakaran and Dr. P. Sengottuvelan

Abstract--- In recent years, there has been significant interest in fault-localization techniques that are based on statistical analysis of program constructs executed by passing and failing executions. This paper shows how an existing techniques such as Tarantula, Jaccard, Ochiai have an efficient technique for localizing fault based on suspiciousness score and rank compared to set-union, set-intersection. A survey of all the existing techniques should be analyzed and a faulty statement in each should be calculated. Each and every technique same test case is generated and an executed statement should be denoted for finding faulty statement.

Keywords--- Tarantula, Jaccard, Set-union, Set-Intersection, Nearest Neighbor, Fault Localization

I. INTRODUCTION

DEBUGGING software is an expensive and mostly manual process. Of all debugging activities, locating the faults, or fault localization, is the most expensive [1]. This expense occurs in both the time and the cost required finding the fault. Because of this high cost, any improvement in the process of finding faults can greatly decrease the cost of debugging.

In practice, software developers locate faults in their programs using a highly involved, manual process. This process usually begins when the developers run the program with a test case (or test suite) and observe failures in the program. The developers then choose a particular failed test case to run, and iteratively place breakpoints using a symbolic debugger, observe the state until an erroneous state is reached, and backtrack until the Fault is found. This process can be quite time-consuming.

To reduce the time required locating faults, and thus the expense of debugging, researchers have investigated ways of helping to automate this process of searching for faults. Some existing techniques use Coverage information provided by test suites to compute likely faulty statements.

Although comparing the fault localization of techniques is difficult, several recent studies have compared four existing techniques in terms of their ability to localize faults. Renieris and Reiss [2] presented their technique, called Nearest Neighbor, and compared it to two techniques that use set union and intersection operations on coverage data. Their studies show that, on a given set of subjects, the Nearest-Neighbor technique performs more effectively than the two set-based approaches.

II. FAULT LOCALIZATION TECHNIQUES

In this section, we present an overview of the fault localization techniques we studied Tarantula, Set union, Set intersection, Nearest Neighbor, Jaccard, Ochiai. For each technique, we describe (1) its method for computing an initial set of suspicious statements in the program; i.e. the set of statements where the search for the fault should begin and (2) its method of ordering (or ranking) the rest of the statements for continuing the search in case the fault is not found in this initial set of suspicious statements.

2.1 Tarantula

Software testers often gather large amounts of data about a software system under test. These data can be used to demonstrate the exhaustiveness of the testing, and find areas of the source code not executed by the test suite, thus prompting the need for additional test cases. These data can also provide information that can be useful for fault localization.

Tarantula utilizes such information that is readily available from standard testing tools: the pass/fail information about each test case, the entities that were executed by each test case (e.g., statements, branches, methods), and the source code for the program under test. The intuition behind Tarantula is that entities in a program that are primarily executed by failed test cases are more likely to be faulty than those that are primarily executed by passed test cases. Unlike most previous techniques that used coverage information (e.g., [3]), Tarantula allows some tolerance for the fault to be occasionally executed by passed test cases. We have found that this tolerance often provides for more effective fault localization.

In particular, the hue of a statement, s, is computed by the following equation:

In Equation 1, passed(s) is the number of passed test cases that executed statement s one or more times. Similarly, failed(s) is the number of failed test cases that executed statement s one or more times. Total passed and total failed are the total numbers of test cases that pass and fail, respectively, in the entire test suite. Note that if any of the denominators evaluate to zero, we assign zero to that fraction.

The hue(s) varies from 0 to 1 | 0 is the most suspicious and 1 is the least suspicious. To express this in a more intuitive manner where the value increases with the suspiciousness,
we can either subtract it from 1, or can equivalently replace the numerator with the ratio of the failed test cases for s. Also, note that we can define this metric for other coverage entities such as branches, functions, or classes.

To illustrate how the Tarantula technique works, we provide a simple example program, mid(), and test suite, given in Figure 1. Program mid() takes three integers as input and outputs the median value. The program contains a fault on line 7 [this line should read 'm = x;'] to the right of each line of code is a set of six test cases: their input is shown at the top of each column, their coverage is shown by the multiplication symbol, and their pass/fail status is shown at the bottom of the columns. To the right of the test case columns are two columns labeled 'suspiciousness' and 'rank.' The suspiciousness column shows the suspiciousness score that the technique computes for each statement. The ranking column shows the maximum number of statements that would have to be examined if that statement were the last statement of that particular suspiciousness level chosen for examination. The ranking is ordered on the suspiciousness, from the greatest score to the least score.

Consider statement 1, which is executed by all six test cases containing both passing and failing test cases. The Tarantula technique assigns statement 1 a suspiciousness score of 0.5 because one failed test case executes it out of a total of one failing test case in the test suite (giving a ratio of 1), and five passed test cases execute it out of a total of five passing test cases in the test suite (giving a ratio of 1). Using the suspiciousness equation specified in Equation 2, we get \( \text{susp}(1) = \frac{1}{1} + \frac{1}{1} \), or 0.5. When Tarantula orders the statements according to suspiciousness, statement 7 is the only statement in the initial set of statements for the programmer to inspect. If the fault were not at line 7, she would continue their search by looking at the statements at the next ranks. There are three statements that have higher suspiciousness values than statement 1. However, because there are four statements that have a suspiciousness value of 0.5, Tarantula assigns every statement with that suspiciousness value a rank of 7 (3 statements examined before, and a maximum of 4 more to get to statement 1). Note that the faulty statement 7 is ranked first; this means that programmer would find the fault at the first statement that they examined.

2.2 Set-Union

It computes the union of all statements executed by passing test cases and subtracts these from the set of statements executed by a failing test case. The resulting set contains the suspicious statements that the programmer should explore first.

That is, given a set of passing test cases p containing individual passed test cases pi, and a single failing test cases f, the set of coverage entities executed by p is Ep, and the coverage entities executed by f is Ef. The set union model gives,

\[
E_{\text{Initial}} = E_f - \bigcup_{p \in P} E_p
\]

Calculate the suspiciousness statement for an example program as given in tarantula technique using set union technique as follows:

Case (1):

In this case we are localizing faults by considering the passed test cases from 2 to 6 but neglecting 1 and a failed test case to find a suspiciousness statement.
Passed Test Cases
1, 2, 3
The entities are \{1, 2, 3, 4, 5, 13\} consider as Set A
3, 2, 1
The entities are \{1, 2, 3, 8, 9, 10, 13\} consider as Set B
5, 5, 5
The entities are \{1, 2, 3, 8, 9, 11, 13\} consider as Set C
5, 3, 4
The entities are \{1, 2, 3, 4, 6, 13\} consider as Set D
Failed Test Case
2, 1, 3
The entities are \{1, 2, 3, 4, 6, 7, 13\} consider as Set E

According to set-union technique, we are computing as
\[ E_{\text{initial}} = \{E\} - \{A \cup B \cup C \cup D\} \]                       \hspace{1cm} (3)
\[ \{A \cup B \cup C \cup D\} = \{1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 13\} \] \hspace{1cm} (4)
\[ \{E\} = \{1, 2, 3, 4, 6, 7, 13\} \] \hspace{1cm} (5)
Substitute equations (4) & (5) in (3), we get
\[ E_{\text{initial}} = \emptyset \]

By using Set-union fault localization techniques we have to find the suspiciousness statement as 7 as calculated similarly in the tarantula. The programmer should examined these statement first out of all the total statements.

Case (2):
Considering all the passed test cases in an example program with a failed test case to find a suspiciousness statement.

Passed Test Cases
\[ F = \{1, 2, 3, 4, 6, 7, 13\} \]
\[ A = \{1, 2, 3, 4, 5, 13\} \]
\[ B = \{1, 2, 3, 8, 9, 10, 13\} \]
\[ C = \{1, 2, 3, 8, 9, 11, 13\} \]
\[ D = \{1, 2, 3, 4, 6, 13\} \]
\[ F \cup A \cup B \cup C \cup D = \{1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 13\} \] \hspace{1cm} (6)
Failed Test Case
\[ E = \{1, 2, 3, 4, 6, 7, 13\} \] \hspace{1cm} (7)
\[ E_{\text{initial}} = \{E\} - \{F \cup A \cup B \cup C \cup D\} \] \hspace{1cm} (8)
Substitute equations (6) & (7) in (8), we get
\[ E_{\text{initial}} = \emptyset \]

If we consider all the passed test cases with a failed test case; we are getting empty or null set it indicates no fault that occur in a sample example program. Comparing the results of case (1) & case (2), we are getting different suspiciousness statements.

2.3 Set-Intersection

The Set-intersection technique computes the set difference between the set of statements that are executed by every passed test case and the set of statements that are executed by a single failing test case. A set of statements is obtained by intersecting the set of statements executed by all passed test cases and removing the set of statements executed by the failed test case. Using the same notation as expressed in equation (2), the set-intersection model gives,
\[ E_{\text{initial}} = \bigcap_{p \in P} E_p - E_f \] \hspace{1cm} (9)

Case (1):
In this case we are localizing faults by considering the passed test cases from 2 to 6 neglecting 1 and a failed test case to find a suspiciousness statement.

Passed Test Cases
\[ A = \{1, 2, 3, 4, 5, 13\} \]
\[ B = \{1, 2, 3, 8, 9, 10, 13\} \]
\[ C = \{1, 2, 3, 8, 9, 11, 13\} \]
\[ D = \{1, 2, 3, 4, 6, 13\} \]
\[ \text{AnB} = \{1, 2, 3, 13\} \] \hspace{1cm} (10)
\[ \text{CnD} = \{1, 2, 3, 13\} \] \hspace{1cm} (11)
Substitute equations (10) & (11) as
\[ (\text{AnB}) \cap (\text{CnD}) = \{1, 2, 3, 13\} \] \hspace{1cm} (12)
Failed Test Case
\[ E = \{1, 2, 3, 4, 6, 7, 13\} \] \hspace{1cm} (13)
Substitute the equations (12) & (13), we get
\[ E_{\text{initial}} = \{(\text{AnB}) \cap (\text{CnD})\} - \{E\} \]
\[ E_{\text{initial}} = \emptyset \]

In the above case shows that there are no faults that occurred in an example program.

Case (2):
Considering all the passed test cases in an example program with a failed test case to calculate suspiciousness statement.

Passed Test Cases
\[ F = \{1, 2, 3, 4, 6, 7, 13\} \]
\[ A = \{1, 2, 3, 4, 5, 13\} \]
\[ B = \{1, 2, 3, 8, 9, 10, 13\} \]
\[ C = \{1, 2, 3, 8, 9, 11, 13\} \]
\[ D = \{1, 2, 3, 4, 6, 13\} \]
\[ \text{FnA} = \{1, 2, 3, 4, 13\} \] \hspace{1cm} (14)
\[ \text{BnC} = \{1, 2, 3, 8, 9, 13\} \] \hspace{1cm} (15)
\[ \text{((FnA) \cap (BnC))} = \{1, 2, 3, 13\} \]
Substitute equations (14) & (15), we get
\[ ((\text{FnA}) \cap (\text{BnC}) \cap D) = \{1, 2, 3, 13\} \]
Failed Test Case
\[ E = \{1, 2, 3, 4, 6, 7, 13\} \]

Substitute these above values, we get
\[ \text{Einitial} = \{((\text{FnA}) \cap (\text{BnC}) \cap \text{D}) \} - \{\text{E}\} \]
\[ \text{Einitial} = \{1,2,3,13\} - \{1,2,3,4,6,7,13\} \]
\[ \text{Einitial} = \{\} \]

In the above case also shows that there are no faults that occurred in an example program. Generally, the fault which occurs in a statement 7 but both the cases in this techniques shows no fault is to be occurred.

<table>
<thead>
<tr>
<th>Line Number</th>
<th>Passed Test Case (5,3,4)</th>
<th>Passed Test Case (5,3,4)</th>
<th>Failed Test Cases (2,1,3)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>*</td>
<td>*</td>
<td>*</td>
</tr>
<tr>
<td>2</td>
<td>*</td>
<td>*</td>
<td>*</td>
</tr>
<tr>
<td>3</td>
<td>*</td>
<td>*</td>
<td>*</td>
</tr>
<tr>
<td>4</td>
<td>*</td>
<td>*</td>
<td>*</td>
</tr>
<tr>
<td>5</td>
<td>*</td>
<td>*</td>
<td>*</td>
</tr>
<tr>
<td>6</td>
<td>*</td>
<td>*</td>
<td>*</td>
</tr>
<tr>
<td>7</td>
<td>*</td>
<td>*</td>
<td>*</td>
</tr>
<tr>
<td>8</td>
<td>*</td>
<td>*</td>
<td>*</td>
</tr>
<tr>
<td>9</td>
<td>*</td>
<td>*</td>
<td>*</td>
</tr>
<tr>
<td>10</td>
<td>*</td>
<td>*</td>
<td>*</td>
</tr>
<tr>
<td>11</td>
<td>*</td>
<td>*</td>
<td>*</td>
</tr>
<tr>
<td>12</td>
<td>*</td>
<td>*</td>
<td>*</td>
</tr>
<tr>
<td>13</td>
<td>*</td>
<td>*</td>
<td>*</td>
</tr>
</tbody>
</table>
Case (1):
In this Case, we are considering a single failed test case statement execution with a passed test case (3, 3, and 5) that is most similar to a failed test case as shown in the above table.

Nearest Neighbor=Failed Test Case-Passed Test Case
={1,2,3,4,6,7,13}-{1,2,3,4,6,7,13}
={}
It indicates no faults that found in an example program.

Case (2):
In this Case, we are considering a single failed test Case with a passed test Case (5, 3, and 4) that is most similar to the failed test case as shown in the above table.

Nearest Neighbor=Failed Test Case-Passed Test Case
={1,2,3,4,6,7,13}-{1,2,3,4,6,13}
={7}
It indicates that the fault that were found in a statement 7.

2.5 Jaccard Technique
In the field of data clustering, similarity coefficients have been proposed that can also be used to calculate suspiciousness ratings. These include the Jaccard [5] coefficient used in the pinpoint program [6]:

\[ S_{\text{Jaccard}}(l) = \frac{\text{Passed}(l)}{\text{Total Passed} + \text{Failed}(l)}, \]

Where Passed \((l)\) is the number of passing executions that execute statement \(l\), Failed \((l)\) is the number of failing executions that execute statement \(l\), Total Passed is the total number of passing test cases, and Total Failed is the total number of failing test cases.

We are calculating Jaccard coefficients for a sample program as given in the tarantula technique, the test case should be similar as in tarantula but the suspiciousness score vary because of Jaccard coefficients and same rank for each and every statements. Here also the statement 7 is a faulty statement as tabulated below.

2.6 Ochiai Technique
Ochiai [7] coefficient used in the molecular biology domain as,

\[ S_{\text{Ochiai}}(l) = \frac{\text{Failed}(l)}{\sqrt{\text{Failed}(l) + \text{Passed}(l)}}, \]

Where Passed \((l)\) is the number of passing executions that execute statement \(l\), Failed \((l)\) is the number of failing executions that execute statement \(l\), Total Passed is the total number of passing test cases, and Total Failed is the total number of failing test cases.

In this technique also we are calculating suspiciousness for each and every statement in a program based on passed and failed test case using Ochiai coefficients. Here also the rank for every statement be similar but differ in their suspiciousness score shown in the below tabulation.

III. CONCLUSION
In the survey of above fault localization techniques, set-union and set-intersection have models to calculate the faulty statement number but it doesn’t show suspiciousness statement score and rank. As in the Tarantula, Jaccard,, Ochiai techniques shows that the statements with same rank for the programmer to examined the faulty statement but differ in their suspiciousness score.

REFERENCES