Identifying Coincidental Correctness for Fault Localization by Clustering Test Cases

Yi Miao\textsuperscript{1,2}, Zhenyu Chen\textsuperscript{1,2}, Sihan Li\textsuperscript{1,2}, Zhihong Zhao\textsuperscript{1,2}, Yuming Zhou\textsuperscript{1}
\textsuperscript{1} State Key Laboratory for Novel Software Technology, Nanjing University, Nanjing, China
\textsuperscript{2} Software Institute, Nanjing University, Nanjing, China
zhaozh@software.nju.edu.cn

Abstract—Coverage-based fault localization techniques leverage coverage information to identify the faulty elements of a program. However, these techniques can be adversely affected by coincidental correctness, which occurs when faulty elements are executed but no failure is triggered. This paper proposes a clustering-based strategy to identify coincidental correctness. The key rationale behind this strategy is that test cases in the same cluster have similar behaviors. Therefore, a passed test case in a cluster, which contains failed test cases, is highly possible to be coincidentally correct. Our experimental results show that, by cleaning or relabeling these possibly coincidentally correct test cases, the effectiveness of coverage-based fault localization techniques can be effectively improved.

Keywords: coincidental correctness; cluster analysis; fault localization

1 INTRODUCTION

Coverage-Based Fault Localization (CBFL) leverages the execution information of both the failed test cases and passed test cases to assist the developer in identifying the program elements that induce a given failure. The intuition behind these techniques 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 [1]. Although CBFL has shown promising results in previous studies, it is still necessary to further improve its effectiveness. One of the main challenges is the coincidental correctness problem.

Coincidental correctness occurs when a test case executes the faulty elements but no failure is triggered. The PIE model presented in J.M. Voas et al., [2] emphasizes that for a failure to be observed, the following three conditions must be satisfied: “Execution”, “Infection”, and “Propagation”. The case is termed weak coincidental correctness, if the program produces the correct output when only the condition “Execution” is satisfied. The case is termed strong coincidental correctness, if the program produces the correct output when only the conditions “Execution” and “Infection” are satisfied. As the second condition of the PIE model has nothing to do with CBFL, strong coincidental correctness is not within the discussion of this paper. Hence, hereafter, coincidental correctness will refer to the former definition as it is more relevant to the execution profiles, and thus has more impact on CBFL.

Previous studies have demonstrated that coincidental correctness is prevalent in both its forms: strong and weak [3], [4]. W. Masri et al. have proven that coincidental correctness is responsible for reducing the safety of CBFL. More specifically, when coincidentally correct test cases are present, the faulty elements will likely be ranked as less suspicious than when they are not present. As shown in the previous studies [4], [5], the efficiency and accuracy of CBFL can be improved by cleaning the coincidentally correct test cases. However, it is difficult to identify coincidental correctness because we do not know the locations of faulty elements in advance.

In this paper, we propose a clustering-based strategy to identify the subset of test suite that is possible to be coincidentally correct. We apply cluster analysis to group test cases into different clusters. According to [6] and [7], test cases in the same clusters have similar behaviors. As such, a passed test case in a cluster, which contains failed test cases, is highly possible to be coincidentally correct because it has the potential to cover the faulty elements as those failed test cases do. We also present two strategies to deal with the coincidental correct test cases (see Section 3 for detail). By cleaning or relabeling these test cases, adverse effects of coincidental correctness can be reduced, which may lead to an increase of the efficiency and accuracy of coverage-based fault localization techniques. The experimental results show that our strategy is effective to alleviate the coincidental correctness problem and to improve the effectiveness of fault localization.

The remainder of this paper is organized as follows. Section 2 details the motivation of our work. Section 3 describes our approach to identify coincidentally correct test cases in detail. Section 4 presents the experimental work and results for the proposed strategy. Section 5 introduces some related work on coincidental correctness and cluster analysis on software testing. Finally, Section 6 presents our conclusions and future work.

2 MOTIVATION

2.1 Prevalence of Coincidental Correctness

In [3], Masri et al. demonstrated that coincidental correctness is prevalent in both its forms (strong and weak). Furthermore, the exhibited levels of weak coincidental correctness are much more significant than those of the strong one. To show the prevalence of the scenario under study, we conducted an experiment on the Siemens programs. The
Siemens set contains seven C programs, and all of them can be downloaded from the SIR repository [17]. Each of the programs has a correct version, a number of faulty versions seeded with a single fault, and a corresponding test suite. We compare the output results of the correct versions with that of the corresponding seeded versions to determine the failures. Failures determined in this manner are called output-based failures. According to the execution profile, if a faulty element is executed during a test case, but no output-based failure is detected, we categorize the test case as coincidentally correct.

Our study only takes into account 115 seeded versions, and excludes the other versions because they contain code-missing errors or the faulty statements are not executable. Figure 1 summarizes the result. It illustrates the exhibited level of coincidental correctness is significant. The horizontal axis represents the percentages of coincidentally correct tests (each bar corresponds to a range of size 10%). The vertical axis represents the percentage of seeded versions that exhibit coincidental correctness. The horizontal axis summarizes the result. It illustrates the exhibited level of coincidental correctness is common in software testing.

2.2 Safety Reducing Effect

Denmat et al. [15] pointed out the limitation of CBFL and argued that the effectiveness of this technique largely depended on the hypothesis that executing the faulty statements leads most of the time to a failure.

In the following, we use Ochiai as an example to show that coincidental correctness is a potential safety-reducing factor. As shown in L. Naish et al., [16], the suspiciousness metric of Ochiai is defined as:

\[ M(e) = \frac{a_{ef}}{\sqrt{(a_{ef} + a_{ef})(a_{ef} + a_{ep})}} \]

where:
- \( e \) = faulty program element
- \( a_{ef} \) = number of failed runs that execute \( e \)
- \( a_{ep} \) = number of failed runs that do not execute \( e \)

Assume that there are \( k \) tests which execute \( e \) but do not raise a failure. Two strategies can be applied on these tests to improve the accuracy of the CBFL technique. The first strategy is to remove these tests from the test suite, that is, to subtract \( k \) from \( a_{ep} \). Consequently, the suspiciousness metric will be:

\[ M'(e) = \frac{a_{ef}}{\sqrt{(a_{ef} + a_{ef})(a_{ef} + a_{ep} - k)}} \]

It is easy to know that \( M(e) \leq M'(e) \). To verify:

\[ M'(e) \geq 0, M(e) \geq 0 \text{ and } M'(e)/M(e) \geq 1 \Rightarrow M(e) \leq M'(e) \]

The second strategy is to relabel those tests from “passed” to “failed”, i.e., to subtract \( k \) from \( a_{ef} \) and add it to \( a_{ep} \), the suspiciousness metric will be:

\[ M''(e) = \frac{a_{ef} + k}{\sqrt{(a_{ef} + a_{ef} + k)(a_{ef} + a_{ep})}} \]

It is easy to know that \( M(e) \leq M''(e) \). To verify:

\[ M''(e) - M'(e) \geq 0 \Rightarrow M(e) \leq M''(e) \]

It can be seen that ignoring coincidentally correct test cases will leads to an underestimating of the suspiciousness of the faulty element.

3 METHODOLOGY

3.1 General Process

Some symbols we use throughout the rest of the paper are explained as follows:

- \( T \): the test suite used for a given program.
- \( T_p \): the set of passed test cases.
- \( T_f \): the set of failed test cases.
- \( T_{cc} \): the set of coincidentally correct test cases.
- \( T_{icc} \): the set of identified coincidentally correct tests.

Given a test suite \( T \), which is comprised of \( T_p \) and \( T_f \), the goal is to identify \( T_{cc} \) from \( T_p \). The result is \( T_{cc} \), and each element of \( T_{cc} \) is a potential candidate of the members of \( T_{cc} \).

In this paper, we propose a clustering-based strategy to obtain \( T_{cc} \). The goal of cluster analysis is to partition objects into clusters such that objects with similar attributes are placed in the same cluster, while objects with dissimilar attributes are placed in different clusters [7]. So, execution profiles are used as the features fed to a clustering algorithm. Specifically, test cases which execute the faulty elements and have similar execution profiles with the failed test cases are likely to be clustered together. Therefore, if a cluster consists of both failed test cases and passed test cases, the passed test cases within this cluster are very likely to be coincidentally correct. Note that our approach is based on the single-fault assumption. Multi-fault programs are not within the discussion of this paper, but will be explored in the near future.

For a developer to find the fault with the help of automatic fault-localization techniques, he/she can use the following procedure to take advantage of our strategy to improve the effectiveness of the diagnosis: First, a set of test cases is executed on the given program. As a result, each test case is labeled "passed" or "failed" according to its output result. Execution profiles which reveal the coverage information are...
collected at the same time. Then, clustering is conducted on the execution profiles. The next step is to identify coincidentally correct test cases using the method mentioned above, and the identified test cases are added to $T_{icc}$. Two strategies (cleaning or relabeling) can be employed to handle with the test cases that belong to $T_{icc}$, see Section 3.2.3 for details. Finally, a CBFL technique is applied to the refined test suite.

3.2 Detailed Technologies

3.2.1 Execution Profile Collection

We use geov (GNU call-coverage profiler) [18] to obtain statement coverage information. For a test case, its statement coverage profile $p_i = <e_1, e_2, ..., e_n>$, where $n$ represents the number of lines of the given program, and $e_i = 1$ if the $i$th line of code is executed, otherwise $e_i = 0$.

3.2.2 Cluster Analysis

The execution profiles of the test suite $T$ are collected to form the input to the cluster analysis. The execution profile of each test case is regarded as an object to be clustered. The number of objects is equal to the number of test cases of $T$. In our context, n-dimensional Euclidean distance [19] is used as the distance function because it is easily calculated and widely used.

The simple K-means is employed as the clustering algorithm in our approach. We chose this algorithm because it is simple and fast. Besides, it performs reasonably well in our previous studies [6], [8]. It takes the number of clusters as a parameter. In our context, this number is set according to the size of $T$. Let $CN$ denote the number of clusters, $CN = |T|^p$, where $|T|$ is the number of test cases in $T$ and $0 < p < 1$.

Note that it is hard to decide a fixed ratio of the number of clusters. It mainly depends on the size of the test suite, and how much risk the developers are willing to take in order to identify the coincidental correct test cases. If the size of test suite is large, a relatively low value of $p$ can be chosen to keep the level of the false negatives. If the developers have a high demand for accuracy of the recognition of coincidental correctness, a relatively high value of $p$ can be chosen to keep the level of false positives.

3.2.3 Handling with the coincidental correctness

Passed test cases which are grouped into the same cluster with the failed ones are very likely to be coincidentally correct, and are added to $T_{icc}$. The reasons are two-fold:

1) A test case which executes the faulty statement does not necessarily induce a failure, but not vice versa. It is a sufficient condition for a failed test case to execute the faulty statements.

2) It is assumed that test cases with similar execution profiles will be clustered together. Therefore, the identified passed test cases will have similar execution profiles with the failed ones.

As such, the identified test cases have a great chance to execute the faulty elements, but still produce the correct output. In other words, they conform to the definition of coincidental correctness.

To deal with $T_{icc}$, we propose two strategies:

- The Cleaning Strategy: Test cases in $T_{icc}$ are removed from the original test suite $T$. According to the suspiciousness metric, it will improve the suspiciousness values of the faulty statements by subtracting the number of coincidental correctness from the number of passed test cases.

- The Relabeling Strategy: The labels of test cases in $T_{icc}$ are changed from "passed" to "failed". It will also improve the suspiciousness values of the faulty statements by subtracting the number of coincidental correctness from the passed test cases and adding it to the number of failed ones. The improvement may be more significant than the cleaning strategy to a certain extent, but it may have risks.

3.2.4 Fault Localization

In this study, we select Ochiai (rather than Tarantula) as the CBFL technique. The main reason is that Ochiai is a recently proposed technique and the metrics is more effective than Tarantula in locating faults. Although these two techniques share the basic principle of CBFL, and they operate on exactly the same input data, as demonstrated in R. Abreu et al., [9], the Ochiai similarity coefficient can improve diagnostic accuracy over other coefficients, including those used by the Pinpoint and Tarantula tools [16]. As a result, it will be more convincing that if our approach can improve a well-performed CBFL technique.

Note that the objective of this study is not to compare various fault localizers, but rather to develop a strategy that will improve the CBFL across multiple fault localizers. Although existing CBFL techniques use different metrics for the coverage entities, most of them share the same basic principle to locate the faulty elements. In other words, they have the same hypothesis and share similar input data. Therefore, we believe that if our approach works on Ochiai, it will perform reasonably well on other fault localizers. In the future work, we will conduct experiments to investigate this conjecture.

4 EXPERIMENT AND EVALUATION

4.1 Subject Programs

The Siemens set is used as the subject programs in our study because it is a particularly widely used benchmark for evaluating software testing and fault localization techniques. The detailed information on these programs is listed in Table 1. The second column of the table shows for each program the number of faulty versions used in our experiment. The third column shows for each program the lines of code (the number of executable statements in the parentheses) that it contains.

<table>
<thead>
<tr>
<th>Program</th>
<th>Versions</th>
<th>LOC (Executable)</th>
<th>Test Suite Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>replace</td>
<td>29</td>
<td>563(243)</td>
<td>5542</td>
</tr>
<tr>
<td>println</td>
<td>3</td>
<td>563(190)</td>
<td>4130</td>
</tr>
<tr>
<td>printtokens2</td>
<td>6</td>
<td>510(200)</td>
<td>4115</td>
</tr>
<tr>
<td>schedule</td>
<td>8</td>
<td>412(150)</td>
<td>2650</td>
</tr>
<tr>
<td>schedule2</td>
<td>4</td>
<td>307(127)</td>
<td>2710</td>
</tr>
<tr>
<td>totinfo</td>
<td>16</td>
<td>406(123)</td>
<td>1052</td>
</tr>
</tbody>
</table>
The program `tcas` is not included because it is too small for cluster analysis. It has only 173 lines of code, of which 54 are executable statements. Therefore, many test cases may have the same execution profiles. Consequently, the number of the clusters generated is limited, and it is difficult to effectively distinguish test cases in this case.

Additionally, we also exclude some of the remaining versions for the following reasons: these versions have no failures detected by any test case. Besides, similar to the experimental setup of Jones and Harrold [1], executable statements are used instead of LOC. Thus we ignore the versions with modifications in the header files, or modifications in a macro statement started with "#define". Furthermore, versions contain code-missing errors are also excluded. Because CBFL hinges on the assumption that a statement which has been executed by most failed test cases is a good candidate for being faulty, and in other words, if a faulty statement causes a test case to fail, then the test case must have executed that statement. However, this will not hold in certain cases such as code-missing fault. In this situation, there is no so-called faulty statement and CBFL cannot localize the fault exactly [14]. Finally, some versions are omitted because they do not contain any coincidental correctness. In summary, we have excluded 25 faulty versions in total, and use 66 versions for our experiment.

4.2 Evaluation Metrics

Similar to [1], to evaluate the ability of our approach to identify coincidental correctness, we compute metrics to quantify the generated false negatives and false positives. Also, to assess the impact of our approach on the effectiveness of CBFL, we use the T-score reduction as the evaluation metric.

1) Measure of generated false negatives:

\[
f = \frac{|T_{cc} - T_{icc}|}{T_{cc}}
\]  

(1)

This measure assesses whether we have successfully identified all the coincidentally correct test cases. The lower the measure value is, the better the recognition accuracy is.

2) Measure of generated false positives:

\[
h = \frac{|(T_p - T_{cc}) \cap T_{icc}|}{T_{cc} - T_{c}}
\]  

(2)

This measure assesses whether we have mistakenly categorized test cases as coincidentally correct. Similarly, the lower the measure value is, the better the recognition accuracy is.

3) Measure of effectiveness improvement:

\[\Delta TS = TS - TS'
\]

where TS and TS’ represents for the T-score before and after applying our approach respectively.

T-score is widely used in evaluating fault localization techniques [1], [10], [11]. It measures the percentage of code that has been examined in order to find the defect, and is defined as follow:

\[T\text{-score} = \frac{|V_{min} - V_{exa}|}{|V|} \times 100\% \]  

(3)

\[|V|\) refers to the size of the program (lines of the executable statements), and \(|V_{examined}\) refers to the number of statements investigated by the programmer in order to find the defect. The lower the T-score value is, the more effective the method will be. Therefore, a larger \(\Delta TS\) implies a greater improvement.

4.3 Experimental Results

4.3.1 Recognition Accuracy

Table 2 shows the ability of our approach to recognize the coincidental correctness. It takes \(p\) (the ratio of the number of clusters) as a parameter, and \(p = 6\%\). This value is selected according to previous studies [6], [8]. The column named “Range” represents the percentage of false negatives and false positives, and the column of “Versions” represents the percentage of versions that exhibit a given range. "FN" and "FP" are short for "False Negative" and "False Positive" respectively.

From Table 2, the following observations can be made about the recognition accuracy of our approach:

1) 2% versions can recognize more than 90% coincidentally correct test cases
2) 21% versions generate 10%-50% false negatives
3) 33% versions generate 50%-90% false negatives
4) 44% versions fail to recognize most of the coincidentally correct tests
5) Most of the versions, 92%, specifically, generate a small number of false positives, in the range [0%, 10%]

It can be speculated that larger number of clusters yields a higher rate of false negatives but a lower rate of false positives. It is reasonable because the purity of a cluster increases as the number of clusters increases. As a result, some of the coincidentally correct test cases, once put into a cluster with some failed ones, are spread to another cluster full of passed test cases. Therefore, these coincidentally correct test cases will be missed. Similarly, some non-coincidentally correct test cases will be spread to another cluster full of passed test cases so that these test cases will not be mistaken for coincidental correctness.

<table>
<thead>
<tr>
<th>Range</th>
<th>Version%(FN)</th>
<th>Version%(FP)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0%~10%</td>
<td>1.51</td>
<td>92.42</td>
</tr>
<tr>
<td>10%~20%</td>
<td>1.51</td>
<td>3.03</td>
</tr>
<tr>
<td>20%~30%</td>
<td>7.57</td>
<td>4.54</td>
</tr>
<tr>
<td>30%~40%</td>
<td>4.54</td>
<td>0</td>
</tr>
<tr>
<td>40%~50%</td>
<td>7.57</td>
<td>0</td>
</tr>
<tr>
<td>50%~60%</td>
<td>3.03</td>
<td>0</td>
</tr>
<tr>
<td>60%~70%</td>
<td>6.06</td>
<td>0</td>
</tr>
<tr>
<td>70%~80%</td>
<td>9.09</td>
<td>0</td>
</tr>
<tr>
<td>80%~90%</td>
<td>15.15</td>
<td>0</td>
</tr>
<tr>
<td>90%~100%</td>
<td>43.93</td>
<td>0</td>
</tr>
</tbody>
</table>
### 4.3.2 Impact on the Effectiveness of Fault Localization

We use box plots to depict the overall experimental results. It takes \( p \) (the ratio of the number of clusters) as a parameter, and \( p = 6\% \). Each box plot represents the statistics of the T-score reduction for each subject program. The bottom and top of the box are 25th and 75th percentile, respectively. The line within the box denotes the median of the values in the box and the point denotes the mean. The ends of the whiskers represent the minimum and maximum of all the data.

Figure 2 and 3 illustrates the impact of our approach on the effectiveness of CBFL. Figure 2 depicts the results applying the cleaning strategy, and Figure 3 depicts the results applying the relabeling strategy. The ideal situation, where all the coincidentally correct test cases are picked out, with 0\% false negatives and 0\% false positives, is also shown on the figures as a comparison. The bold dot presents the average T-score reduction for each program under the ideal situation.

19.57\% versions have been improved by the cleaning strategy, and the T-score reduction can reach up to 8.55\%. 33.33\% versions have been improved by the relabeling strategy, and the T-score reduction can reach up to 22.0\%.

In summary, as shown in Figure 2 and 3, the cleaning strategy is relatively a safe method, because if \( p \) is set to a reasonable value, 6\% in our case, more than 87\% versions will be improved or stay the same, with an increase rate of 0.67\%~8.55\%, while the rest 13\% may be deteriorated, with the decrease rate of -2.0\% ~ -0.41\%.

The effectiveness of fault localization of some versions remains the same. We observe that the faulty statements in most of the versions have already been ranked at the very front position, such that dealing with the coincidental correctness will have little effect on the improvement. Moreover, as presented in W. Masri et al., [5], although cleaning coincidental correctness will lead to an increment of the suspiciousness metric of the faulty statement, the rank of the statement will not necessarily increase correspondingly. Therefore, the effectiveness of CBFL after applying our methodology remains the same or even gets worse for some versions. The key factors that influence the improvement of CBFL are the rate of false negatives and false positives, which are heavily depended on the clustering results.

Furthermore, we conducted a paired t-test on the differences between the T-scores before and after applying our strategy. The paired t-test is a statistical technique that is used to compare two population means in the case of two samples that are correlated, especially in a “before- after” study. Suppose the T-score before using our strategy is \( A \), and the T-score after using our strategy is \( B \). \( H_0 \): \( A \leq B \). \( H_a \): \( A > B \). Note that, during this test, only T-scores less than 20\% are taken into account. The reason is that it is not common to ask programmers to examine more than 20\% of the code in practice [11]. It can be observed that, by using cleaning strategy and relabeling strategy, the \( p \)-value is 0.01 and 0.02, respectively, both of which implies that the improvement is significant at the 0.05 level.

![Figure 2. Impact of cleaning strategy on the effectiveness of CBFL](image1)

![Figure 3. Impact of relabeling strategy on the effectiveness of CBFL](image2)

4.4 Threats to Validity

The threats to external validity include the use of Siemens set as our subject programs. As a matter of fact, these programs are all small C programs and the faults are manually injected. To reduce this threat, we plan to apply our technique to larger programs in the future.

The threats to internal validity include the tools we use to generate execution profiles and conduct the cluster analysis. In our context, we use gcov to record coverage information and rely on the data mining tool Weka for cluster analysis. Both of them are mature and widely used. Another issue related to internal validity is the clustering algorithm we choose. As in our study, we use simple K-means because it is simple and effective. However, having the failed test cases clustered either too centralized or too scattered will have adverse effect on the
results. To be more specific, they will lead to high false negative and false positive rates, respectively.

5 RELATED WORK
Voas [2] introduced the PIE model, which emphasizes that for a failure to be observed, the following three conditions must be satisfied: “Execution”, “Infection”, and “Propagation”. W. Masri et al. [3] have proved that coincidental correctness is responsible for reducing the safety of CBFL.

As shown in the previous studies [4, 5], the efficiency and accuracy of CBFL can be improved by cleaning the coincidentally correct test cases. However, it is challenging to identify coincidental correctness because we do not know the location of fault beforehand. X. Wang et al. [4] have proposed the concept of context pattern to help coverage refinement so that the correlation between program failures and the coverage of faulty statements can be strengthened. W. Masri et al. [5] have presented variations of a technique that identify the subset of passed test cases that are likely to be coincidentally correct. One of these techniques first identifies program elements (cc_e) that are likely to be correlated with coincidentally correct test cases. Then it categorizes test cases that induced some cc_e as coincidentally correct. The set of coincidentally correct test cases would be partitioned into two clusters further. A more suspicious subset will be cleaned to improve the effectiveness of fault localization. The experimental result is promising, however, although it used the same subject program (the Siemens test suite) as ours in their experiment, it is applicable to only 18 versions of the 132 versions, which has a smaller application scope than our approach (applicable to 66 out of 132 versions).

Previous empirical observations have shown that, by cluster analysis, test cases with similar behaviors could be grouped into the same clusters. Therefore, cluster analysis has been introduced for test case selection. Vangala et al. [12] used program profiles and static execution to compare test cases and applied cluster analysis on them, identifying redundant test cases with high accuracy. Dickinson et al. introduced cluster filtering technique [7, 13]. It groups similar execution profiles into the same clusters and then selects a subset of test cases from each cluster based on a certain sampling strategy. Since test cases in the same cluster have similar behaviors, the subsets are representative for the test suite so that it is able to find most faults by using the selected subsets instead of the whole test suite.

6 CONCLUSIONS AND FUTURE WORK
In this paper, we proposed a clustering-based strategy to identify coincidental correctness from the set of passed test cases. To alleviate the adverse effect of coincidental correctness on the effectiveness of CBFL, two strategies, either removing or relabeling, were introduced to deal with the identified coincidentally correct test cases. We conducted an experiment to evaluate the proposed approach. The experimental results suggested that it achieved approximate results as the ideal situation did.

We intend to conduct more comprehensive empirical studies and explore the following issues in our future work:

1) Search for better clustering algorithms to fit in with this scenario. As denoted in section 4.4, the failed test cases clustered either too centralized or too scattered would lead to poor results. We use K-means in our experiment for its simplicity, and there are many other clustering algorithms need to be explored.

2) Conduct empirical studies on how multiple-faults affect the result of our approach and explore how to deal with this situation to minimize the adverse effects.

REFERENCES