Improved Similarity Computation Method of Execution Trace

Yue Yan*
Chongqing Engineering Laboratory for Detection, Control and Integrated System, Chongqing Technology and Business University, Chongqing, China
College of Computer Science and Information Engineering, Chongqing Technology and Business University, Chongqing, China

ABSTRACT: Fault localization is a crucial point for the quality of software test. After analyzing the advantages and disadvantages of fault localization method based on trace, we proposed a new computation method to confirm the correlation pattern of statement blocks through relevance of minor defects in the program. We also introduced this method into computation of vector similarity and established a new computation method for execution trace similarity. With realization of details as the basis, the effectiveness of this new method has been verified in combination with NN method. Compared with other commonly-used algorithms, this computation method has higher localization when fault rate is low and can maintain better stability when fault rate is high.

Keywords: fault localization; execution trace; similarity degree; suspiciousness ratio; test case

1 INTRODUCTION

Reliability and usability are the key factors in measuring software quality. However, fault hidden by software can directly affect software reliability and usability. How to effectively localize fault is the most critical procedure in removing software fault. Classified by fault, fault localization can be divided into various methods, such as probability program dependence graph [1] and slice spectrum [2] designed from program data dependence or control dependence, fault localization method for execution trace mining based on correlation rules [3][4] and complicated relation between test case and authority processed by neural network [5][6], thought optimization realized by particle swarm [7], and optimization and supplement on software test technology structure from perspective of demand [8]. Although all the methods can improve the traditional debugging efficiency, it is inevitable to think about how to ensure input completeness and solve low test running efficiency issue while applying them.

In software test, it was noticed that execution traces of test cases which didn’t pass were different from those of test cases which passed. Collection of the minimum statement blocks leading to failure can be localized by comparing the execution traces of test cases which can pass and which fail to pass. This method is called fault localization method based on execution trace [9][10]. Both currently-used dynamic fault localization based on program frequency spectrum [11] and code fault localization used to compute fault contribution rate [12] belong to this type. However, all the above methods seem to aim at single fault localization in general. They are lack of consideration on fault ripple efficiency and are hard to carry out. As fault localization of trace execution software can be combined with automated testing, it is of great practical significance to figure out how to maintain high efficiency of software fault localization method.

2 DEFINITION AND ANALYSIS OF FAULT

Set program $P = \{s_1, s_2, \ldots, s_m\}$ in which $s_j$ refers to No. $j$ statement block. Set test case suite $Te=\{Te_1, Te_2, \ldots, Te_n\}$ in which $Te_i=(I_i, O_i)$, $I_i$ refers to input of specified test case while $O_i$ refers to the corresponding expected output. Set $Tr_e$ as an execution trace of test case $Te_i$ in program $P$, and thus it is a sequence of statement block executed by $Te_e$. Obviously, according to testing rules, after all test cases are
executed, test case suite \( Te \) can correspond to an execution trace set \( Tr \), and elements in \( Te \) can correspond to elements in \( Tr \) respectively. Suite \( Tr \) is divided into two mutually disjoint suites: suite \( Trp \) and suite \( Trf \). \( Trp \) is the execution trace set of test cases which can pass while suite \( Trf \) is that of test cases which fail to pass.

From perspective of testing, there may be fault in each statement block. Set \( sus(s_j) \) as the suspiciousness ratio of statement block \( s_j \). A higher suspiciousness ratio means the statement block is very likely to have hidden fault. The aim of software fault localization based on trace is to confirm a reasonable fault localization report according to analysis of suite \( Trp \) and suite \( Trf \). Set \( R=\{s_1, s_2, s_3, ..., s_m\} \) as a sequence formed by suspiciousness ratio of a statement block subset of program \( P \) in which \( sus(s_1) \geq sus(s_2) \geq sus(s_3) \geq ... \geq sus(s_m) \). Programmer will check statement blocks one by one according to fault localization report until he can localize the actual fault. It is evident that how to confirm suspiciousness ratio of each statement block in program \( P \) is the key part in fault localization report quality.

3 ANALYSIS OF EXISTING PROBLEMS IN CURRENT METHODS

There are four phases in current general framework of software fault localization methods based on trace\(^{13}\):

Organization of execution trace: arrange information collected in testing phase, and organize execution traces of test cases in a certain pattern.

Selection of execution trace: select execution trace used for fault localization and eliminate noise in the data to bring more accurate computation of suspiciousness ratio.

Current fault localization methods based on trace usually choose all test cases which fail to pass and one test case which fails to pass. The efficiency and effectiveness of fault localization depends on whether the test cases can truly reflect the characteristics of the tested software, that is select the minimum \( n \) groups of test cases as much as possible and each test case can cover a software fault. However, a test case failing to pass may be caused by several software faults in practical situation. Hence, how to select \( n \) groups of test cases and how to localize fault have always been the main directions in software testing research.

Computation of suspicious ratio: analyze the selected execution traces and compute suspiciousness ratio of each statement block by a certain computational model.

Assessment of localization report: rate localization report by assessment standard and evaluate the effect of fault localization.

In the above framework, description of execution trace is necessary and introduction of vector in theoretical analysis is one of the most common and the most sophisticated methods. For example, when statement block \( s_j \) is executed, \( Tr[\text{index}] = 1 \) or 0. This is a two-value vector definition. Moreover, vector value can also be used to represent how many times a statement block is executed. For example, if statement block \( s_j \) is executed for \( x \) times, the corresponding vector value should be \( x \). If the block has not been executed, the value should be 0. This is a multiple-value vector definition. It is easy to get a \( Tr(\text{exec}) \) by program insert technology in which \( n \) refers to the number of test case and \( m \) refers to the number of statement block. Theoretically, as long as test cases can satisfy enough testing needs, the aim of software testing can definitely be reached by checking \( Tr \). However, it is impossible to check each testing trace \( Tr_i \) in actual fault localization analysis and one fault may exist in several testing traces. It is not necessary to check all traces one by one. Therefore, selecting execution trace according to execution trace distance is the most direct way, that is using distance between execution trances to represent similarity and take it as the basis in computing \( sus(s_j) \). At present, there are many similarity computation methods based on vector.

Relative integrity of program block must be taken into consideration when conducting module division on program \( P \) in an object-oriented program as it is not an equal or random allocation. For example, a program block of some object or some complicated method must maintain relative independence to confirm corresponding \( 1 \) and \( O \). This division can directly affect the generation of execution trace. Moreover, statements contained in many modules have certain relevance from the perspective of software design, such as gathering, composition, and generalization. For example, after a class contains a fault, the fault is very likely to be taken to other subclasses and thus may bring testing traces of related statement blocks into \( Trf \). Furthermore, if a parent class passes the test while its derived class fails to pass, there is no doubt that comparing the execution traces of its derived classes can do great help in fault localization. Nevertheless, the above factors have not been applied to computing methods based on vector trace similarity. Undoubtedly, improving the above trace computing methods based on this idea can be helpful to produce localization reports with higher efficiency.
4 IMPROVED FAULT LOCALIZATION METHOD BASED ON TRACE

While organizing execution trace by vector, Cosine coefficient is often applied to compute the similarity of execution trace \( Tr \) and execution trace \( Tr \). See the equation as follows:

\[
\text{Sim}(Tr, Tr') = \frac{\sum_{i=1}^{m} tr_{ai} tr_{bj}}{\sqrt{\sum_{i=1}^{m} tr_{ki}^2} \sqrt{\sum_{i=1}^{m} tr_{kj}^2}}
\]

(1)

For example, in the execution trace set of program \( P \), there are \( Tr_{i} = (1,1,0,0,0,0,0,0) \), \( Tr_{j} = (0,0,1,1,0,0,0,0) \), and \( Tr_{k} = (1,0,0,0,0,0,1,0) \) defined in two-value vector mode, in which \( Tr_{i} \in Tr_{j} \) and \( Tr_{i} \notin Tr_{j} \). Assume there is fault existing in statement block \( s_{i} \). According to equation (1), the similarities among them are:

\[
\text{sim}_{cos}(Tr_{i}, Tr_{j}) = 0, \quad \text{sim}_{cos}(Tr_{i}, Tr_{j}) = 0.5, \quad \text{sim}_{cos}(Tr_{j}, Tr_{j}) = 0.
\]

Then, according to NN (Nearest Neighbor) method \cite{9}, \( Tr_{i} \) and \( Tr_{j} \) should be selected to compute suspiciousness ratio. The most direct way to compute the execution traces of test cases which can pass and those of test cases which fail to pass for suspiciousness ratio computation is 0-1 method, that is to assume there is no fault contained in the execution traces of test cases which can pass and assume that fault only exists in the execution traces of test cases which fail to pass, and then remove statement blocks executed by test cases which can pass the test from execution traces of test cases which fail to pass. As a result, the left part will be suspicious statement blocks, and thus statement block \( s_{j} \) will be deleted from fault localization report, which means the suspiciousness ratio will be 0. However, if we select \( Tr_{i} \) and \( Tr_{j} \) to compute suspiciousness ratio, the suspiciousness ratio of statement block \( s_{j} \) will be 1. The direct cause of this phenomenon is there exists \( \text{sim}_{cos}(Tr_{i}, Tr_{j}) < \text{sim}_{cos}(Tr_{i}, Tr_{j}) \) while computing execution trace similarity. Hence, \( Tr_{j} \) instead of \( Tr_{j} \) or \( Tr_{j} \) is selected in trace selection. Results will be similar if other suspiciousness ratio computing methods are applied.

In the above case, if the logical relation among statement blocks \( s_{i}, s_{j}, \ldots, s_{m} \) can be found according to software logical relations, and computing method of execution trace can be improved in this way, the above phenomenon can be avoided.

**Definition 1.** \( CRF \) is a forest with \( n \) trees and \( CRF = (CT_{1}, CT_{2}, \ldots, CT_{m}) \)

**Definition 2.** Definition of \( CT_{i} = (CD_{i}, R_{i}) \) is:

\( CD_{i} \neq \emptyset \);

If \( CD_{i} \) only contains one data element \( cs_{i} \), then \( R_{i} \) is a null set and \( cs_{i} \) must can and only can correspond to statement block \( s_{i} \) (\( 0 < i \leq m, m \) refers to the number of statement block) of program \( P \), saying \( cs_{i} \) is \( root_{i} \); or \( R_{i} = \{H\} \), \( H \) is in the binary relation given below:

If \( CD_{r}(\text{root}_{i}) \neq \emptyset \), then there is a division of \( CD_{i} \), \( CD_{2} \ldots, CD_{q} \) (\( q>1 \)) in \( CD_{r}(\text{root}_{i}) \). For any \( j \neq k \) (\( 1 \leq j, k \leq q \)), there exists \( CD_{j} \cap CD_{k} \neq \emptyset \), and for any \( i \neq j \), there exists an unique data element \( cs_{i} \in CD_{i} \) which fits the following relations: \( < \text{root}_{i}, cs_{j} \geq H \).

Corresponding to division of \( CD_{r}(\text{root}_{j}) \), \( H = \{< \text{root}_{1}, cs_{2}, \ldots, \text{root}_{m}, cs_{n}>\} \) has only one division \( H_{1}, H_{2}, \ldots, H_{q} \) (\( q>0 \)). For any \( j \neq k \) (\( 1 \leq j, k \leq q \)), there exists \( H_{1} \cap H_{2} = \emptyset \). And for any \( i \leq j \), \( H_{i} \) is the binary relation in \( CD_{i} \) (\( CD_{i} \), \( \{H, i\} \) is a tree that fits this definition.

Set \( VS \) (when \( CD_{i} \) only has one data element \( cs_{i} \), it should also be included) as the set of all statement block nodes and make consistent numbers according to sequence numbers of corresponding statement blocks, that is \( VS = \{vs_{1}, vs_{2}, \ldots, vs_{m}\} \), and \( vs_{i} \) is in one-to-one correspondence relation to statement block \( s_{i} (0 < i \leq m) \) of program \( P \).

From this, we can see that \( CRF \) should be a forest consisted of several trees and there is no empty tree inside it.

**Algorithm 1.** Construct a forest \( CRF \) with \( n \) trees

The algorithm can generate a forest with \( n \) trees which can accord with Definition 1 and Definition 2. It can be used to assist computation of execution trace similarity based on vector.

1) Construct set \( CRF = \{CT_{1}, CT_{2}, \ldots, CT_{m}\} \) with \( m \) trees according to the nodes \( \{vs_{1}, vs_{2}, \ldots, vs_{m}\} \) represented by the given \( m \) statement blocks, in which every tree \( CT_{i} \) only has one root node and the subtrees beside it are both empty.

2) Investigate the program definition contained in statement block from perspective of software design. If there is generalization relation between a node \( vs_{i} \) and \( vs_{j} \) in \( VS \), there is set membership between \( CT_{i} \) and \( CT_{j} \). If \( vs_{i} \) and \( vs_{j}, \ldots, vs_{l} \) are part of gathering or composition relation, \( CT_{i} \) and \( CT_{j} \) are in brother relation. If the statement block represented by the integrity of \( CT_{i} \) and \( CT_{j}, \ldots, CT_{l} \) is not included in \( VS \), a new node will be generated and it will be in set membership with them. If there is only relevance relation in \( vs_{i}, CT_{i} \) is an independent tree.

3) After all generalization, gathering and composition relations in \( VS \) are all confirmed through steps stated in 2), if the root of the tree where it belongs is not included in \( CRF \), add it to \( CRF \).

4) Repeat procedures stated in 2) and 3) until \( CT_{m} \) is completed.

As classes and their relations of a program are confirmed during software design, it is easy to get \( CRF \) with reference of class diagram or class dependence graph without much additional burden during statement block division. When there is no multiple-successional program, \( CRF \) should be a tree consisted of several trees. The number of tree \( n \) and the size of each depend on the structure and complexity of class definition of the software. In fact, the application of this method is to decide a relation between statement block \( \{s_{i}, s_{j}, \ldots, s_{m}\} \) according to characteristics.
of software, and then provide proof for similarity computation improvement.

Definition 3. Set any two leaf nodes in \( CT_i \) as \( vs_1 \) and \( vs_2 \), and the least common ancestor \( LCA(vs_j, vs_j) \) is the node with the \( \rightarrow \) on the common ancestor path of \( vs_1 \) and \( vs_2 \). When \( vs_j \) and \( vs_j \) don’t belong to the same tree, it is stipulated that \( LCA=\max (\text{depth}_{vs_1}, \text{depth}_{vs_2})+1 \).

In other words, when the relations between classes defined by program of statement block gets closer, their layers in the tree get deeper accordingly. For any two \( vs_i \) and \( vs_j (i \neq j \neq m) \), the definition is as follows:

\[
\frac{2*\text{depth}(LCA(vs_i, vs_j))}{\text{depth}(vs_i) + \text{depth}(vs_j)}
\]

In the above definition, when and only when there is \( vs_i = vs_j \), the value is 1 and the range of similarity is within (0,1] and won’t be 0. With this basis, the computing equation for similarity of execution traces \( Tr_i \) and \( Tr_j \) is given below:

\[
\similarity(Tr_i, Tr_j) = \frac{\rightarrow vs_i \cdot vs_j}{\sqrt{\rightarrow vs_i \cdot vs_i \cdot vs_j \cdot vs_j}}
\]

In the above case, if there is CRF as shown in Figure 1, the similarity degrees computed by Equation (3) are: \( \similarity_{\text{crf}}(Tr_1, Tr_2) = 0.8, \similarity_{\text{crf}}(Tr_1, Tr_3) = 0.65, \similarity_{\text{crf}}(Tr_2, Tr_3) = 0.52 \). According to NN method, 0-1 method should still be applied to compute suspiciousness ratio, then the suspiciousness ratio of statement block \( x_i \) will be 1 which can reach the aim of the thesis assumption.

Figure 1. A part of CRF in a program.

5 METHOD ANALYSIS AND CONCLUSIONS

Obviously, Equation (3) is consistent with definition of traditional vector space modal. Moreover, it can be found through Equation (2) that even if \( i \neq j \), dot product \( vs_i \cdot vs_j \) won’t be 0. That is to say, it differentiates the execution traces with similarities computed by Equation (1) as 0 to a certain degree. In this way, the following computation of suspiciousness ratio can become easier, such as applying NN method to compute suspiciousness ratio.

Then, we can analyze the definition and structure algorithm of CRF. It uses generalization, gathering and composition relations of class diagram or class dependence diagram with no consideration of relevance relation. This method can not only simplify CRF, but can also maintain a very convenience structure. It can even be regarded as affiliated outcome of class diagram or class dependence diagram.

To move forward, we find that the reason for why we can achieve the improvement effect as shown in the above case lies in the relation between the executed statement blocks in \( Tr_i \) and \( Tr_j \). In Figure 1, \( vs_1, vs_2, vs_3, vs_4 \) are shown in brother relation. However, from perspective of software design, the program of the statement blocks represented by them can form a certain class by gathering or composition relation. If \( Tr_i \in Tr_p \) and \( Tr_j \in Tr_t \), it is obviously easier to find fault by comparing \( Tr_i \) and \( Tr_j \). But in \( Tr_i \) and \( Tr_j \), although \( vs_1 \) and \( vs_2 \) are in brother relation, their relations with \( vs_3 \) are much weaker. Therefore, even if the condition remains the same as \( Tr_i \in Tr_p \) and \( Tr_j \in Tr_t \), the effect of finding fault by comparing \( Tr_i \) and \( Tr_j \) will be worse. In other words, for \( Tr_i \in Tr_p \) and \( Tr_j \in Tr_t \), if the executed statement blocks have certain relations in class definition, Equation (3) will improve the originally simple vector similarity computing method. Thus, it has greater universal meaning. Evidently, when \( Tr_i, Tr_j \in Tr_p \) or \( Tr_j \), it is not necessary to discuss whether the similarity can be improved.

We chose Siemens Suits (http://sir.unl.edu/portal/index.php) which is widely-recognized typical data used to assess fault localization technology effectiveness at present. There were 7 groups in total. See Table 1 for the comparison between CRF proposed in this thesis and the commonly-used methods based on trace. Two-element group \((x,y)\) means x test cases which pass the test were selected while y means y test cases which fail to pass were selected. 0-1 model was applied to compute suspiciousness ratio. In accordance with practice [14] we rated localization reports with fault ratio of >20%, >40% and >80% as Good, Medium, and Bad respectively.

<table>
<thead>
<tr>
<th>Method</th>
<th>Dicing</th>
<th>NN-Binary</th>
<th>NN-Integer</th>
<th>CRF</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of execution trace</td>
<td>((x,y))</td>
<td>((1,1))</td>
<td>((1,1))</td>
<td>((x,y))</td>
</tr>
<tr>
<td>Localization effect</td>
<td>bad</td>
<td>medium</td>
<td>good</td>
<td>Good</td>
</tr>
</tbody>
</table>

Based on the above procedures, we extracted 4 groups of C language programs with different functions. There were 41 different fault versions in each group in which each fault version contained a fault manually inserted on purpose. See Table 2 for the comparison between the above method and other commonly-used ones.
Table 2. Comparison of efficiency of various test methods (2).

<table>
<thead>
<tr>
<th></th>
<th>Ochiai</th>
<th>Tarantula</th>
<th>CRF</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fault ratio (%)</td>
<td>20</td>
<td>40</td>
<td>80</td>
</tr>
<tr>
<td>Found fault ratio (%)</td>
<td>25.5</td>
<td>55</td>
<td>84.6</td>
</tr>
</tbody>
</table>

Note: fault ratio refers to the percentage of fault which needs to be checked (function, statement block, etc.) in the whole fault-suspicious statement sequence. Fault-detection ratio refers to the percentage of found fault in all faults.

It can be seen that compared with other commonly-used algorithms, CRF can have better localization when fault rate is low and can still keep good overall performance and better stability when fault rate is high.

6 CONCLUSIONS

The nature of fault localization based on execution trace is to localize fault by comparing and localizing execution traces in $T_r_p$ and $T_r_f$. For $T_r \in T_r_p$, how to find a $T_r \in T_r_f$ which is similar to the greatest extent can directly decide the accuracy of the following suspiciousness ratio computation. This thesis proposed an effective fault localization method and verified its effectiveness. There is still much space in exploring and deepening large-scale data test, usage of earlier software testing experience data, and fault localization assessment method. How to consider interaction among statement blocks during fault localization, how to improve granularity, and study of class level based on fault localization of code level will all be important directions for future software fault localization research.

REFERENCES


