Testing-Based Interactive Fault Localization

Dan Hao
Institute of Software, School of Electronics Engineering and Computer Science, Peking University, Beijing, 100871, P.R. China
haod@sei.pku.edu.cn

Categories and Subject Descriptors
D.2.4 [Software Engineering]: Software/Program Verification;
D.2.5 [Software Engineering]: Testing and Debugging–debugging aids; D.2.7 [Software Engineering]: Distribution, Maintenance, and Enhancement;

General Terms
Performance, Design, Reliability, Human Factors, Verification

Keywords: Debugging, Maintenance, Fault localization

1. PROBLEM DESCRIPTION
As programmers usually can hardly always produce faultless code in the first place, debugging is an inevitable task in the process of software development and maintenance. Actually, debugging has been identified as one of the most time-consuming tasks [7]. The situation may become even worse when a developer or a maintainer has to face code written by others.

Typically, there are mainly two steps in debugging: 1) finding the location of the fault, and 2) replacing the faulty statement(s) with the correct one(s). In the literature, four main tasks in the process of finding the location of the fault have been identified [7]: 1) identifying statements involved in failures; 2) narrowing the search by selecting suspicious statements that might contain faults; 3) hypothesizing about suspicious faults; and 4) restoring program variables to a specific state to verify the hypothesis. In practice, a developer usually focuses on one fault revealing execution, as it can often be assumed that there is at least a faulty statement along the execution trace. Usually, manual fault localization is an interactive process: The developer firstly hypothesizes that all the statements that involve in the execution are suspicious. Then, he or she sets some breakpoints along the fault revealing execution trace and re-runs the execution trace to gather information around the breakpoints. According to this information, he or she can further narrow down the scope of suspicious statements. The process is repeated until he or she finds where the fault is.

To help human debugging, many approaches have been recently proposed to localize faults automatically (see e.g. [1] [2] [3] [4] [5] [6] and [7]). Among them, testing-based fault localization (TBFL) is quite promising. Typically, these testing-based fault localization approaches utilize the information acquired from testing, such as the coverage information and the execution results, to calculate the “suspicion” of each statement and rank the statements according to their suspicion. Thus, the debugger can check the ranking list of the statements from the top to the bottom until he or she finds the fault. Compared with the manual process of fault localization, TBFL approaches are advantageous as they can utilize multiple execution traces, while a debugger can only rely on one execution trace in the manual fault localization process. However, manual fault localization still has its own advantages, as the hypotheses verified or rejected in the previous steps of fault localization can be used in the coming steps in this process.

In my PhD period, my research focuses on investigating techniques that can take both the advantages of interactive fault localization and testing-based fault localization. In my research, I refer to these techniques as testing-based interactive fault localization.

2. PRIOR RESEARCH
In the literature, there are quite a few approaches to fault localization relying on test information, such as dicing [1], TARANTULA [7], Nearest Neighbor Queries [10] and our previous approach SAFL [6], which are based on the analysis of a large number of execution traces to localize the faults. The dicing approach is based on the concept of dices, one of which is subtracting the set of statements executed by a passed test case from those executed by a failed one. The statements are ranked by the number of dices in which they are contained. TARANTULA defines a pair of the color value and the brightness value to rank the statements. The calculation of the two values is also based on the execution traces. The Nearest Neighbor Queries approach compares the spectra of the correct runs and faulty runs to produce a report of “suspicious” parts of the program. SAFL takes the similarity between test cases into consideration. It combines the fuzzy set theory and the probability theory to calculate the suspicion probability of statements. All these approaches only provide a ranked list of statements for programmers to look through one by one. Different from these testing base fault localization approaches, Delta Debugging approach [13] [14] focuses on identifying the portion of a failed test case ultimately to cause the fault and this approach is demonstrated in [2] to be effective for finding the failure-causing state. However, with these
approaches, debuggers typically have to check about 10% (see the experimental results in [6], [10] and [2]) of the all the lines of source code to find the fault. Obviously, these results are not satisfactory enough for a large program. Thus debuggers can benefit little from such approaches.

Similar to above approaches, [11] proposes an approach for spreadsheet applications, which measured the suspicious of the cells. However, it only applies to end-users.

Liblit et al instrumented predicates in the source code to gather data from user executions, and used the date to rank the predicates on their conditional probability to further confirm the location of the bugs [8] [9]. So its result is still a ranked list, which is not convenient for programmers for future use.

3. PROPOSED SOLUTION
The basic idea of our approach is depicted as Figure 1. The whole approach is divided into three phases: pre-processing testing information, constructing a suspicion model, and setting checking points and gathering human debugging information. In our approach, checking points have a similar meaning with breakpoints. The first phase is to execute the faulty program with the test suite and record the execution information as a matrix between the test cases and the statements, as well as the execution results. The second phase is to use the testing information to construct the suspicion model.

![Figure 1. Overview of Our Approach](image)

The third phase is to set checking points according to the suspicion model and gather more information about the program from human estimation and execution of the program, which is used to further adjust the suspicion model. This phase composes two steps:

1) Setting checking points. We choose and set one or more checking points according to the suspicion model.
2) Examining the checking points and gathering feedback information. Debuggers estimate whether the state at the checking point is correct or not. If correct, the suspicion of the statements executed before the checking point will be decreased while that of the statements executed after the checking point will be increased. Otherwise, the change of suspicion of statements should be the contrary. This is the feedback information acquired from this step, and it should be reflected in the suspicion model.

With the gathered feedback information from the third phase, the suspicion model can be reconstructed. Then the second and the third phases will be repeated until the approach can confine the suspicious statements to a scope small enough for debuggers to find the fault.

It is obviously that the first phase is initialization. The second and the third phases comprise the iterative fault localization. Phase two constructs a suspicion model to determine the checking points for phase three and phase three collects information to further refine the suspicion model obtained in phase two.

4. EXPECTED CONTRIBUTIONS
The expected contributions of my dissertation are:
1) It will provide the debuggers with an aid in setting breakpoints. Thus, debuggers do not have to consider where to set the breakpoints especially when they are facing a large program.
2) It accelerates the process of fault localization. The approach automatically calculates and chooses the checking points for debuggers. More information gathered from the human debugging process is used to refine the calculation of the suspicion of statements.

5. CURRENT PROGRESS
5.1 Work to Date
We have proposed to use the fuzzy set theory for constructing the initial suspicion model and it has been applied to our previous approach to testing-based fault localization named SAFL [6].

Intuitively, a test case can be viewed as a set of statements which contribute to its result (failed or passed). It is uncertain to judge to what extent a statement belongs to the sets for test cases executing the statement because of its fuzzy contribution.

In the conventional set theory, whether an element belongs to a set is deterministic, which is a yes-or-no question. However, real conditions are often not deterministic and cannot be described precisely. The fuzzy set theory is firstly proposed to generalize the classical notion of a set and use a proposition to accommodate such kind of fuzziness [15]. It defines a concept for an element, named as the “grade of membership”, which represents the grade of the element belonging to a set.

Thus, we can represent a test case as a fuzzy set, which is composed of statements, each labeled with a grade of membership.
After executing the program, which is composed of statements \{s_1, s_2, ..., s_n\}, with test case suite \( T = \{t_1, t_2, ..., t_p\} \), we will have the execution matrix representing which statement is executed by each test case in matrix \( E = (e_{ij}), 1 \leq i \leq n, 1 \leq j \leq m \) where
\[
e_{ij} = \begin{cases} 1, & s_j \text{ is executed by } t_i \ (1 \leq j \leq m) \\ 1, & t_i \text{ is passed} \quad (j = m + 1) \\ 0, & \text{otherwise} \end{cases} \tag{1}
\]
Because of the fuzzy membership of a statement, one test case can be deemed as equal. Thus, the relation between test cases and their executed statements can be denoted as a test case can be viewed as a fuzzy set composed of the statements which are involved in its execution trace. Initially, the contributions of the statements to the result (failed or passed) of the test case can be deemed as equal. Thus, the relation between test cases and their executed statements can be denoted as a fuzzy matrix \( F = (f_{ij}), 1 \leq i \leq n, 1 \leq j \leq m \) where
\[
f_{ij} = \begin{cases} 1 / \sum_{j=1}^{m} e_{kj}, & \text{if } e_{ij} = 1 \\ 0, & \text{otherwise} \end{cases} \tag{2}
\]
Based on formula 2 and the fuzzy set theory, the suspicion of each statement is as follows
\[
P(j) = \frac{\sum_{k=1}^{m} \max(f_{ik} \mid e_{ij} > 0 \land e_{i(m+1)} = 0 \land 1 \leq i \leq n)}{\sum_{k=1}^{m} \max(f_{ik} \mid e_{ij} > 0 \land 1 \leq i \leq n)} \tag{3}
\]

Based on this technique, when using the feedback information to reconstruct the suspicion model, we can change the membership of the statements based on the information. This change of membership can be naturally reflected in the new suspicion model.

5.2 Future Work

We plan to further work on the following two aspects in the future. First, we will further improve our suspicion model on the membership. In our previous work, we assume that all the executed statements contribute equally to the passed or failed result, which is only a simplified way. So we will take more information into consideration to determine the membership grade of different statements. Second, we will refine our approach, taking the more information into consideration. The detailed plan is as follows.

1) If debuggers estimate the statement at the checking point and determine that it is correct. Then at this time it is known that this statement is not associated with the failed test case and its contribution to the failure is zero. Then the membership grade of it to the involving test cases is changed to be zero and that of the other executed statements to the same test case change simultaneously. With this information as feedback, the suspicion model thus can be refined.

2) The execution times of each statement will also be considered. Given the execution times of each statement before setting checking points and after setting checking points, we can further reduce the scope of suspicious statements.

3) As what we wanted in our previous work was whether the output is right or wrong, the exact values of output were gained as a seemingly useless byproduct. We can use symbolic analysis on these output variables. However, its effectiveness needs further evaluation.

4) Statements are not independent. Most of them are associated in many ways, such as control dependence and data dependence. For example, in conditional statement “if A then B”, statement B is control dependent on statement A, and only if statement A is executed, statement B then has the possibility to be executed. Therefore, if the statement B, which is constrained by the condition A, is checked to be correct, then the suspicion of the statement A, which contains the condition, will also be reduced. We would like to give high priority to such statement, which is associated with many statements. Because if such statement is checked to be correct, the priority of many associated statements will further be refined, and some associated statements may even be reasoned to be correct.

5) We can use the slicing technique to achieve further refinement. For example, the debugging information between two executions with different checking points injected can be used to narrow the search space. There may exist two executions, one checking point of which is checked to be correct, while the other is wrong. Assume the former checking point is \( C_1 \), the latter is \( C_2 \). And the corresponding forward slice for \( C_1 \) is \( Sc_1 \), while the backward slice for \( C_2 \) is \( Sc_2 \). If \( C_2 \) depends partially on \( C_1 \), then \( C_2 \) will be in the forward slice \( Sc_1 \), the suspicious statements are reduced to those executed between the two checking points and in the intersection of forward slice \( Sc_1 \) and backward slice \( Sc_2 \). Otherwise, if \( C_2 \) does not depend on \( C_1 \), we can further choose a checking point in the program located between checking point \( C_1 \) and \( C_2 \).

6. PLAN FOR EVALUATION

The proposed approach will be evaluated on the Siemens Test Suite [2] and some practical programs such as Space [12], Tiny C Compiler, DC and Count. Siemens Test Suite is a set of seven faulty C programs; each program is ranged from 138LOC to 516LOC. For each program, there are several faulty versions and a test suite containing several thousand test cases. The Space program is a larger program in C, which is developed for the European Space Agency, whose LOC is 9,564. For the eight programs, including Siemens Test Suite and the Space program, each faulty version only contains one fault. To evaluate whether our approach is effective for programs with multi-faults, we collect the faults in different faulty programs and inject them in one program. Then we can gather faulty programs with multi-faults. The last three programs are also C programs ranged from 11,000 LOC to 512 LOC, which have been used in [5] [6]. So the faulty versions and test cases of these programs are also available.
We will use RECON (please refer to http://www.cs.uwf.edu/~recon/recon3/index.html ) to track the execution traces of the programs. RECON is a tool that can collect path traces from C program execution.

We will count how many checking points have been checked before we find the fault with the help of our approach, how many breakpoints have been set for human debugging without help and how many statements have been checked for current fault localization approaches. We use these as the measure to evaluate our approach, human debugging process and current fault localization approaches. Additionally, Renieris in [10] proposed an evaluation framework for fault localization approaches. We plan to perform our evaluation for the current fault localization approaches and our newly proposed one. Besides the evaluation on effectiveness, we will explore the time cost of our approach. As it is an iterative approach, the time cost is also a main concern.

For current fault localization approaches, people have to check the statements on the ranked list one by one. In order to estimate whether the statement is correct or wrong, he may have to check the result after the statement and check the results before the statement and after the statement. However our approach will check the result after the statement and estimate whether the statement is right or wrong. In many cases, the human involvement in them may not be comparable. So, we also plan to perform some case studies to evaluate the approaches in real debugging environment. We plan to select some Master students in our laboratory to use current fault localization approaches and our interactive one to find real faults respectively, and then base our evaluation on the students' subjective comments on the effectiveness of these approaches for debugging real world programs.

Furthermore, we also plan to evaluate our approach on other programs in different programming languages, such as JAVA, C++. This can help to generalize our results acquired from the above evaluation.

7. ACKNOWLEDGMENTS
This effort is sponsored by the National 973 Key Basic Research and Development Program No. 2002CB312003, the State 863 High-Tech Program No. 2005AA113030, and the National Science Foundation of China No. 60403015. I am grateful to my advisor, Professor Jiasu Sun, and Associate Professor Lu Zhang, for their guidance and support.

8. REFERENCES