Effective Fault Localization using Code Coverage

W. Eric Wong[1], Yu Qi[1], Lei Zhao[1], and Kai-Yuan Cai[2]

[1] Department of Computer Science, University of Texas at Dallas, USA
Email: {ewong,yxq014100, lxz064000}@utdallas.edu
[2]Department of Automatic Control, Beijing University of Aeronautics and Astronautics, China, Email: kycai@buaa.edu.cn

Abstract

Localizing a bug in a program can be a complex and time-consuming process. In this paper we propose a code coverage-based fault localization method to prioritize suspicious code in terms of its likelihood of containing program bugs. Code with a higher risk should be examined before that with a lower risk, as the former is more suspicious (i.e., more likely to contain program bugs) than the latter. We also answer a very important question: How can each additional test case that executes the program successfully help locate program bugs? We propose that with respect to a piece of code, the aid introduced by the first successful test that executes it in computing its likelihood of containing a bug is larger than or equal to that of the second successful test that executes it, which is larger than or equal to that of the third successful test that executes it, etc. A tool, χDebug, was implemented to automate the computation of the risk of the code and the subsequent prioritization of suspicious code for locating program bugs. A case study using the Siemens suite was also conducted. Data collected from our study support the proposal described above. They also indicate that our method (in particular Heuristics III (c), (d), and (e)) can effectively reduce the search domain for locating program bugs.

Keywords: fault localization, program debugging, code coverage, risk of code, successful tests, failed tests

1. Introduction

A common practice for testing a software application[1] in industry is to execute a large suite of test cases all at once in batch mode and record any failures detected during the testing in a log file. The failure information is then used by programmers to debug the software and locate program bugs. A possible scenario is that for a large test suite, say 1000 test cases, a majority of them, say 995, are successful test cases and only a small number of failed test cases (five in this example) will cause an execution failure. The challenge is how to use these five failed tests and the 995 successful tests to conduct an effective debugging.

One way to debug a program when it shows some abnormal behavior is to analyze its memory dump. Another way is to insert *print* statements around suspicious code to print out the values of some variables. Neither of these techniques is effective because the former might require an analysis of a tremendous amount of data and the latter puts the burden on the programmers to decide whether these *print* statements should be inserted. A better strategy is to let the system tell you the possible locations of the bugs. We can take advantage of how a program is executed by the failed and the successful tests to prioritize its source code based on the likelihood of containing bugs. Code with a higher priority should be examined first rather than that with a lower priority, as the former is more suspicious than the latter, i.e., more likely to contain the bugs.

Execution slice-based fault localization methods have been explored in several studies where an execution slice with respect to a given test case contains the set of code executed by this test. In Reference [4], a very simple heuristic using an execution dice obtained by subtracting the execution slice of one successful test case from that of one failed test is examined for its effectiveness in locating program bugs. The tool χSlice, as part of the Telcordia's Visualization and Analysis Tool Suite (formerly χSuds), is developed to support this heuristic [3,20]. In Reference [19], more sophisticated debugging heuristics using multiple successful and multiple failed tests are applied to software architectural design in SDL (a specification and description language [6]). In Reference [18], execution slice and inter-block data dependency-based heuristics are used to debug C programs. In Reference [17], we report how execution slice-based fault localization methods are used in an industrial setting for Java applications. However, none of these studies properly answer the question discussed above because they all treat the contribution of each successful test case to program debugging the same.

To overcome this problem, we propose a code coverage-based fault localization method. For a given piece of code (e.g., a statement), we first identify how many successful and failed tests execute it. Then, we want to explore whether all the successful test executions provide the same contribution to aid in program debugging. Intuitively, the answer should be "no." Our proposal is that if a piece of code has already been executed successfully 994 times, then the contribution of the 995th successful execution is likely to be less than the contribution of the second successful execution when the code is only executed successfully once. Stated differently, for a given piece of code, we propose that the aid introduced by the first successful test that executes it in computing its likelihood of containing a bug is larger than or equal to that of the second successful test that executes it, which is larger than or equal to that of the third successful test that executes it, and so on. Different heuristics (refer to Section 2 for more details) are used to adjust the contribution of each successful test execution.

To demonstrate the feasibility of using our proposed method for locating program bugs, a case study on the seven programs in the Siemens suite [7] is conducted. A tool (χDebug) to support this process is also implemented as it is not only time-consuming but also mistake-prone to manually prioritize code based on its likelihood of containing program bugs. Results from our study suggest that Heuristics III (c), (d), and (e) can perform better than other methods in locating

[1] In this paper, we use "software," "application," and "program" interchangeably. We also use "bugs," "faults," and "defects" interchangeably.

31st Annual International Computer Software and Applications Conference(COMPSAC 2007)
0-7695-2870-8/07 $25.00 © 2007 IEEE

program bugs, as they lead to the examination of a smaller percentage of the code before the bugs are located.

The rest of the paper is organized as follows. Section 2 describes our methodology. A case study in which our method is applied to the Siemens suite is presented in Section 3. Also included is an introduction to our debugging tool -- χDebug. Section 4 discusses a few related issues. Section 5 gives an overview of some related studies. Finally, in Section 6 we offer our conclusions and recommendations for future research.

2. Methodology

Given a piece of code (a statement, say $\mathcal{S}$, in our case) and a specific bug, say $\mathcal{B}$, suppose we want to determine how likely it is that $\mathcal{S}$ contains the bug $\mathcal{B}$. Without losing generality, let us assume $\mathcal{S}$ is executed by m successful test cases and n failed test cases.[2] Below we present three heuristics to show how such information can be used to prioritize source code in terms of its likelihood of containing a program bug.

Heuristic I:

If the program execution fails on a test case, it is natural to assume that, except for some special scenarios which will be discussed in Section 4, the corresponding bug resides in the set of code executed by the failed test. In addition, if a statement is executed by two failed tests, our intuition suggests that this statement is more likely to contain the bug than a statement which is executed only by one failed test case. If we also assume every failed test case that executes $\mathcal{S}$ provides the same contribution in program debugging, then we have the likelihood that $\mathcal{S}$ contains the bug is *proportional* to the number of failed tests that execute it. In this way, we define the risk of the j^{th} statement as

$$\sum_{i=1}^{n+m} C_{i,j} \times r_i \quad \text{where}$$

$$\begin{cases} C_{i,j} = 1, & \text{if the } i^{th} \text{ test case executes the } j^{th} \text{ statement} \\ C_{i,j} = 0, & \text{otherwise} \\ r_i = 1, & \text{if the program execution on the } i^{th} \text{ test case fails} \\ r_i = 0, & \text{otherwise} \end{cases} \tag{1}$$

Let's use the example in Figure 1 to demonstrate how Heuristic I computes the risk of each statement. The left matrix shows how each statement is executed by each test case and the right matrix indicates that program execution fails on test cases t_1, t_3, t_{10}, t_{15}, and t_{18}. To compute the risk of $\mathcal{S}_{10}$, we find that it is executed by all five failed tests. From Equation (1), the risk of $\mathcal{S}_{10}$ equals $\sum_{i=1}^{20}(C_{i,10} \times r_i)=5$. Similarly, $\mathcal{S}_1$, $\mathcal{S}_8$ and $\mathcal{S}_9$ also have the same risk of 5 because each of them is executed by five failed tests, whereas $\mathcal{S}_5$ has a risk of 4 as it is executed by four failed tests, and $\mathcal{S}_3$, $\mathcal{S}_6$, and $\mathcal{S}_7$ have a risk of 3 for three failed executions. As for $\mathcal{S}_2$ and $\mathcal{S}_4$, their risk is zero because they are not executed by any failed test.

Heuristic II:

In Heuristic I, we only take advantage of failed test cases to compute the risk of a statement. However, we also observe that if a statement is executed by a successful test case, its likelihood of containing a bug is reduced. Moreover, the more successful tests that execute a statement, the less likely that it contains a bug. If we also assume every successful test case that executes $\mathcal{S}$ provides the same contribution in program debugging, then we have the likelihood that $\mathcal{S}$ contains the bug is *inversely proportional* to the number of successful tests that execute it. Combined with our observation on the failed tests, we define the risk of the j^{th} statement as

$$\sum_{i=1}^{n+m} C_{i,j} \times r_i \; - \; \sum_{i=1}^{n+m} C_{i,j} \times (1 - r_i) \quad \text{where} \tag{2}$$

$$\begin{cases} C_{i,j} = 1, & \text{if the } i^{th} \text{ test case executes the } j^{th} \text{ statement} \\ C_{i,j} = 0, & \text{if the } i^{th} \text{ test case does not executes the } j^{th} \text{ statement} \\ r_i = 1, & \text{if the program execution on the } i^{th} \text{ test case fails} \\ r_i = 0, & \text{if the program execution on the } i^{th} \text{ test case succeeds} \end{cases}$$

This gives the risk of each statement as equal to the number of failed tests that execute it minus the number of successful tests that execute it.

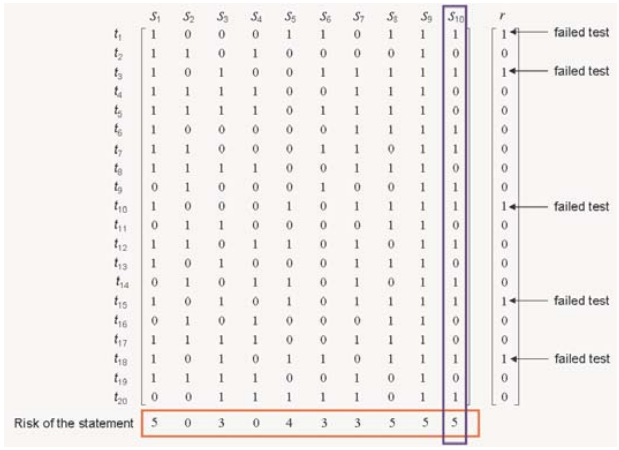

Figure 1. An example of Heuristic I

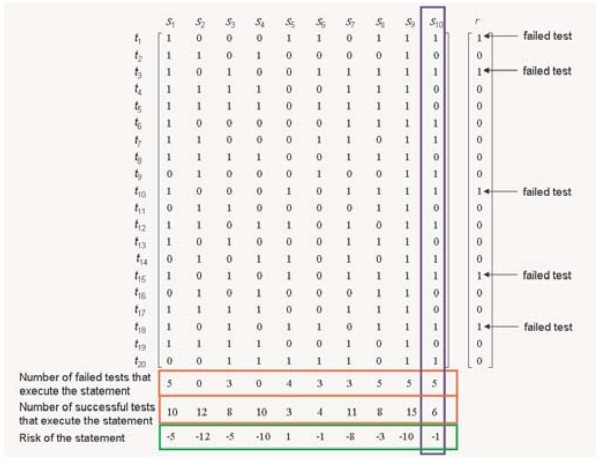

Figure 2. An example of Heuristic II

Let us use the same matrices in Figure 1 to demonstrate how Heuristic II computes the risk of each statement. From Equation (2), the risk of $\mathcal{S}_{10}$ equals $\sum_{i=1}^{20}C_{i,10}\times r_i - \sum_{i=1}^{20}C_{i,10}\times(1-r_i)=5-6=-1$. This is consistent with the fact that $\mathcal{S}_{10}$ is executed by five

[2] All these n failed tests are with respect to the same program bug $\mathcal{B}$. If this is not the case, we need to partition the failed tests into different groups for different bugs. When we do the debugging for a specific bug, we should only use the group of failed tests related to that bug.

0-7695-2870-8/07 $25.00

failed tests and six successful tests. Similarly, S_6 also has the same risk of -1 because it is executed by three failed tests and four successful tests. Refer to Figure 2 for the risk of other statements.

Heuristic III:

In the first two heuristics, we make no distinction between the contributions from different successful test cases. However, as explained in the Introduction, if S has already been executed by many successful test cases, the contribution of an additional successful test case to the risk of S is likely to be less than the contribution of the first few successful test cases. Hence, we propose that for a given statement S, the aid introduced by the first successful test that executes it in computing its likelihood of containing a bug is larger than or equal to that of the second successful test that executes it, which is larger than or equal to that of the third successful test that executes it, and so on. With this in mind, we can, for example, define the first two successful test cases to give a contribution of -1 each, the 3rd to the 10th successful test cases to give a contribution of -0.1 each, and all subsequent successful test cases to each give a very small contribution -α where α can be 0.001, 0.0001, etc. If so, we can define the risk of the j^{th} statement as

$$\sum_{i=1}^{n+m} C_{i,j} \times r_i \; - f\left(\sum_{i=1}^{n+m} C_{i,j} \times (1 - r_i)\right) \text{ where} \qquad (3)$$

$$\begin{cases} C_{i,j} = 1, & \text{if the } i^{th} \text{ test case executes the } j^{th} \text{ statement} \\ C_{i,j} = 0, & \text{if the } i^{th} \text{ test case does not executes the } j^{th} \text{ statement} \\ r_i = 1, & \text{if the program execution on the } i^{th} \text{ test case fails} \\ r_i = 0, & \text{if the program execution on the } i^{th} \text{ test case succeeds} \end{cases}$$

$$\text{and } f(k) = \begin{cases} k, & \text{for } k = 0, 1, 2 \\ 2 + (k - 2) \times 0.1, & \text{for } 3 \le k \le 10 \\ 2.8 + (k - 10) \times \alpha, & \text{for } k \ge 11 \end{cases}$$

where α is a small number. The key idea of Heuristic III is to use a function to adjust the contribution of each additional successful test in computing the risk of code it executes, even though the exact formula (such as whether the contribution is -0.1 for the 3rd to the 10th successful tests or -0.2 for the 5th to the 11th successful tests) may vary according to the program being debugged. The general guideline is to partition all the successful test cases into three groups. One group contains the first few successful tests, each of which provides a large contribution. Another group contains the next few successful tests, each of which provides a medium contribution. The final group contains the remaining successful tests, each of which only provides a small contribution. In our case, Equation (3) is a sample formula used for programs in the Siemens suite. Refer to Section 3 for more details.

Once again, let us use the same matrices in Figure 1 to demonstrate how Heuristic III computes the risk of each statement. Let $\alpha = 0.01$ in Equation (3). For S_{10}, it is executed by five failed tests each of which gives a contribution of 1. It is also executed by six successful tests with each of the first two giving a contribution of -1 and each of the remaining four giving a contribution of -0.1. As a result, the total contribution from the six successful tests is -2.4. Combined with the contribution of the failed tests, S_{10} has a risk of 5 - 2.4 = 2.6 which is consistent with the risk computed by Equation (3) (namely, $\sum_{i=1}^{20} C_{i,10} \times r_i \; - f\left(\sum_{i=1}^{20} C_{i,10} \times (1 - r_i)\right) = 5 - f(6) = 5 - (2 + (6 - 2) \times 0.1) = 5 - 2.4 = 2.6$). Similarly, since S_7 is executed by three failed tests and eleven successful tests, its risk equals $3 - f(11) = 3 - (2.8 + (11 - 10) \times 0.01) = 3 - 2.81 = 0.19$. Refer to Figure 3 for the risk of other statements.

Our next step is to apply these heuristics to the programs in the Siemens suite to examine how suspicious code can be identified to help us locate bugs effectively.

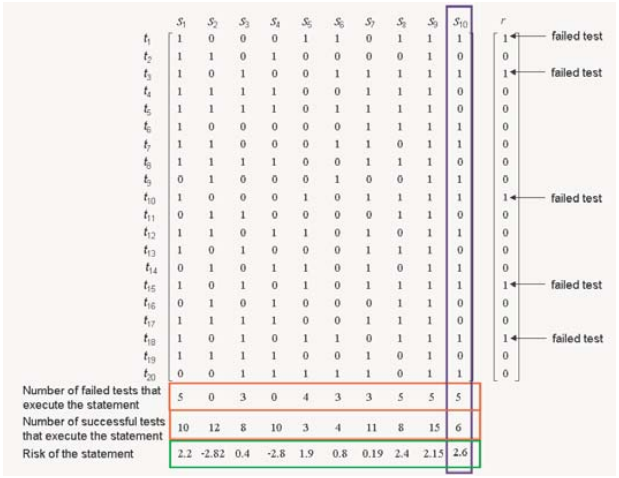

Figure 3. An example of Heuristic III with $\alpha = 0.01$

3. A Case Study

We first provide an overview of the programs, test cases and defects used in our study. A debugging tool, χDebug, is explained next. Then, we present our results and analysis based on these results. The objective of this study is to demonstrate the feasibility of using the method described in Section 2 for locating bugs and to compare our results with those reported in other studies.

3.1 Programs, test cases, and defects

All the programs (correct and faulty versions) and test cases are downloaded from a web site at Georgia Tech [15]. The same set of data has been used by other fault localization studies [5,9,14]. This makes the comparison between our results and their results much easier. Table 1 gives the name and a brief description of each program, the number of faulty versions, LOC, and the number of functions and test cases. These programs are implemented in the C language. Each faulty version contains exactly one fault. Each fault may span multiple statements which may not be contiguous. Refer to [7] for a more detailed description of the programs, versions, and test cases.

Since the web site at Georgia Tech did not provide the execution result for each test case, we had to rerun all the tests to determine whether each execution failed or succeeded. This was done by comparing the result of each faulty version with the expected result of the corresponding correct version. Special attention was required to examine execution failures that were the result of segmentation faults. We also had to measure the coverage of each test case, as such information was not available at the web site either. In summary, we instrumented the programs using χSuds [20], reran all the tests on a Sun server running SunOS 5.9, collected the coverage information, identified which statement was executed by which test(s), and determined whether each execution failed or succeeded.

Program	No. of Fault Versions	LOC†	No. of Test Cases	No. of Functions	Description
print_tokens	7	344	4130	20	Lexical analyzer
print_tokens2	10	355	4115	21	Lexical analyzer
schedule	9	292	2650	18	Priority scheduler
schedule2	10	262	2710	16	Priority scheduler
replace	32	512	5542	21	Pattern replacement
tcas	41	135	1608	8	Altitude separation
tot_info	23	273	1052	16	Information measure

† LOC gives the number of lines of code (i.e., statements) after removing comments and blank lines

Table 1. The seven programs in the Siemens suite

Of the 132 faulty versions, eleven (instead of ten as reported in [9]) were not used in our study. Versions 4 and 6 of "print_tokens" differ from the correct versions only in a *header* file but not in any C file. Version 10 of "print_tokens2," version 32 of "replace," and version 9 of "schedule2" were removed because no test cases fail. Versions 19 and 27 of "replace," and versions 1, 5, 6, and 9 of "schedule" were removed because χSuds does not save all the coverage data before the program crashes. A similar problem was also reported in [9] when gcc with gcov was used to collect the coverage data. Another point worth noting is that the number of test cases downloaded from Georgia Tech (i.e., the number of test cases used in our study) is slightly larger than the number of test cases reported in [9]. Reference [9] does not provide any explanation of why some of the test cases posted at the web site were removed from their study.

3.2 χDebug: a debugging tool based on code coverage

To automate the fault localization, including the computation of the risk of each statement and the highlighting of the suspicious statement(s), we need to have tool support. With this in mind, we developed χDebug: a debugging tool that supports the method as discussed in Section 2.

Given a program and its successful and failed test cases, χDebug displays the source code and its corresponding risk in a user-friendly GUI as shown in Figure 4.[3] Each executable statement is highlighted in an appropriate color with its risk listed to the left of the statement. The numerical range associated with each color in the spectrum at the top gives the risk range represented by that color. The most suspicious code, i.e., the code with the highest risk, is highlighted in red, and the non-executable statements are not highlighted in any color. Depending on the program displayed, different numbers of colors (up to seven) with different numerical ranges associated with each color are automatically decided by χDebug.

In addition to the graphical user interface allowing programmers to visualize suspicious code for possible bug locations, χDebug also provides a command-line text interface via shell scripts for a controlled experiment like the one reported here. In the latter, the bug locations are already known before the experiment, and the objective is not to *find* the location of a bug but to *verify* whether the bug is in the suspicious code identified by using our method. Appropriate routines can be invoked directly to compute the risk of each statement using the heuristics discussed in Section 2 and check whether such suspicious code indeed contains the bug(s). The percentage of code that has to be examined before a given bug is detected is also computed. It is this approach which makes χDebug very robust for our case study.

[3] Since suspicious code is highlighted in different colors based on its risk, it is better to view Figure 4 in color. The same also applies to other figures (Figure 5 to Figure 11) because curves in these figures are displayed in different colors.

3.3 Results and Analysis

During the test execution, χSuds can record which statement is executed by which test case and how many times the statement is executed. If a statement is executed at least once by a test case, we say that the statement is covered by that test case. Otherwise, the statement is not covered by that test case. In our study, all the comments, blank lines, non-executable statements (e.g., function and variable declarations) are excluded for analysis.

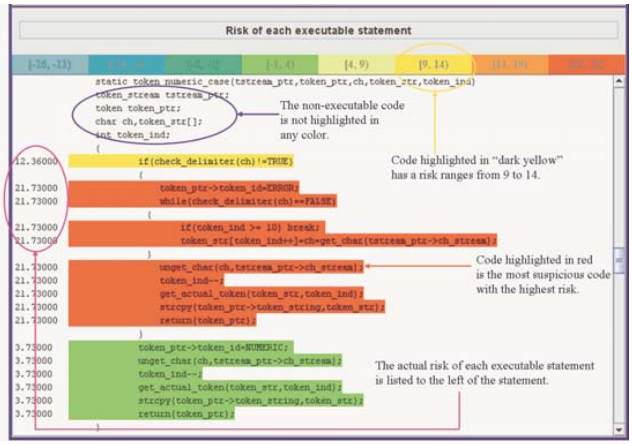

Figure 4. Visualizing suspicious code for possible bug locations

For each executable statement, its risk is initialized to zero and recomputed based on how many successful and failed tests execute the statement and which heuristic is used. All the executable statements are then ranked in descending order according to their risk. The statement at the top has a larger risk than the statement at the bottom; that is, the statement at the top is more likely to contain program bugs than the statement at the bottom. To find a bug, we examine the program starting from the statements at the top until we reach the statement that contains the bug. If a bug spans multiple statements, the examination stops when the first statement containing the bug is reached. For comparison purposes, we compute a score (as defined in [9]) for each faulty version as the percentage of program (executable statements in our case) that need not be examined to find the bug. Hence, if we have to examine 5% of the executable statements for a faulty version, the score of this faulty version is 95%. Following the convention used in [5,9,14], the effectiveness of a technique is pictorially displayed by a curve where the horizontal axis represents the percentage of the program that need not be examined, and the vertical axis represents the cumulative percentage of the faulty versions that achieve a score higher than each segment listed. Each segment is 10 percentage points, except for the first two segments which are 99%-100% and 90%-99%, respectively.

From Reference [9], we observe that for the seven programs in the Siemens suite, the Tarantula technique is more effective in fault localization than the set-union, set intersection, nearest-neighbor, and cause-transitions techniques [5,14]. Refer to Section 5 for a description of these techniques. To avoid having too many curves in each figure which reduces its readability significantly, we only focus on the comparison between the effectiveness of the Tarantula technique and that of our method. If we can show that our method is more

effective in fault localization than the Tarantula technique, then our method is also more effective than the set-union, set intersection, nearest-neighbor, and cause-transitions techniques.

Heuristic I versus Tarantula

The comparison between the effectiveness of Heuristic I and Tarantula is shown in Figure 5. The green curve (labeled as "Tarantula (from [9])) is for the Tarantula technique using the data in [9]. However, as explained in Section 3.1, the failed and successful tests are re-identified by rerunning each test case, and the coverage is re-measured by using χSuds. Also, the number of faulty versions and the number of test cases in our study are slightly different from the ones in [9]. To make a fair comparison, we re-compute the effectiveness of Tarantula using their ranking strategy and our data. The two curves labeled as "Tarantula Best (using our data)" and "Tarantula Worst (using our data)" give the best and the worst effectiveness, whereas the "best" and the "worst" assume that the faulty statement is the first and the last statement, respectively, to be examined among all the statements of the same rank. For example, suppose there are ten statements of the same rank of which one is faulty. The best effectiveness is obtained by assuming the faulty statement is the first to be examined among these ten statements, whereas the worst effectiveness is obtained by assuming the faulty statement is the last to be examined. The two curves labeled as "Heuristic I Best" and "Heuristic I Worst" give the best and the worst effectiveness of our method using Heuristic I. In the rest of the paper we focus on the comparison between the effectiveness of our method and "Tarantula Best (using our data)" and "Tarantula Worst (using our data)"

From Figure 5 we observe that the best effectiveness of Heuristic I is better than the best effectiveness of Tarantula, but its worst effectiveness is worse than the worst effectiveness of Tarantula. In addition, the gap between the best and the worst curves of Heuristic I is too large. This implies that the actual effectiveness of Heuristic I can be any value between "Heuristic I Best" and "Heuristic I Worst" which is very unstable. Therefore, Heuristic I is not a very good strategy for localizing program bugs.

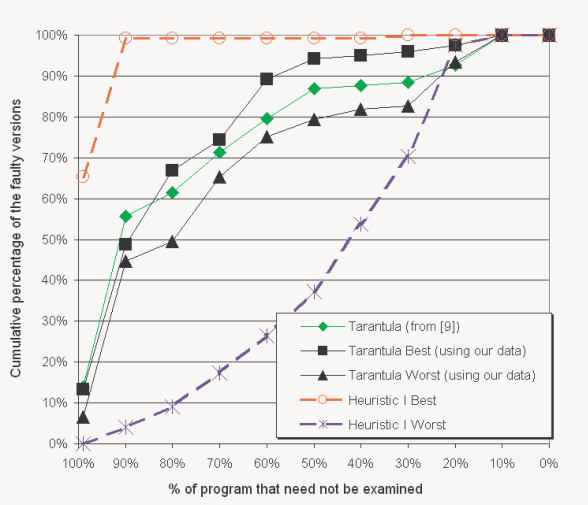

Figure 5. Comparison between Heuristic I and Tarantula

Heuristic II versus Tarantula

Figure 6 gives the comparison between Heuristic II and Tarantula. All the legends are the same as in Figure 5. We observe that the gap between "Heuristic II Best" and "Heuristic II Worst" is much smaller when compared with the gap in Figure 5. However, the best effectiveness of Heuristic II decreases significantly although the worst effectiveness is slightly improved. The reason for such a decrease is because for each faulty version the number of successful tests is much larger than the number of failed tests. As a result, the contributions of these successful tests in computing the risk of the code they execute disproportionately outweigh the contribution of the failed tests. For example, let us refer to version 2 of the program "tot_info." It has a faulty statement which is executed by ten failed tests but 953 successful tests. According to Heuristic II, the risk of this faulty statement is 10 – 953 = -943 which makes it incorrectly marked as a very unsuspicious statement. To overcome this problem, we need to reduce the contribution of each additional successful test. A possible solution is to use Heuristic III.

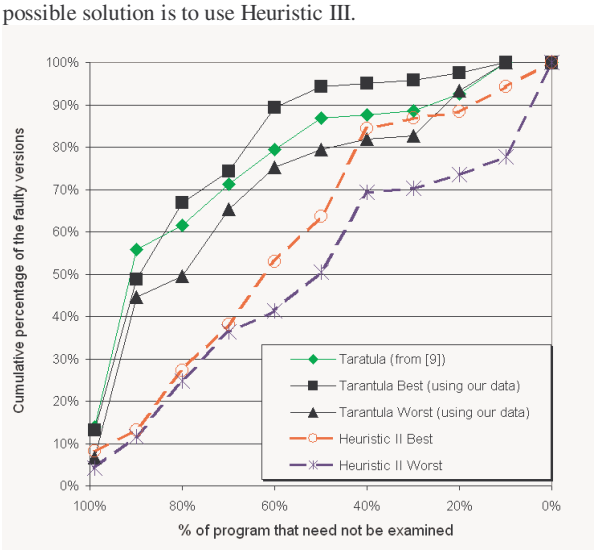

Figure 6. Comparison between Heuristic II and Tarantula

Heuristic III (a) versus Tarantula with α = 0.01

Figure 7 gives a comparison between Heuristic III (a) with α = 0.01 and Tarantula. All the legends are the same as in Figure 5. In this heuristic, for the first two successful tests and the first two failed tests, they give the "same" contribution (except that the former is negative and the latter is positive). Starting from the 3rd successful test, the contribution of each additional successful test is one-tenth of the contribution of each additional failed test. And, starting from the 11th successful test, the contribution of each additional successful test is one-hundredth of the contribution of each additional failed test (because α = 0.01).

We can easily observe that the effectiveness of Heuristic III (a) is a significant improvement. In addition, when compared with Heuristic I, its gap between the best effectiveness and the worst effectiveness is also reduced significantly. In some cases "Heuristic III (a) Best" is better than "Tarantula Best," while in other cases it is the reverse. When compared with "Tarantula Worst," "Heuristic III (a) Worst" is in general better.

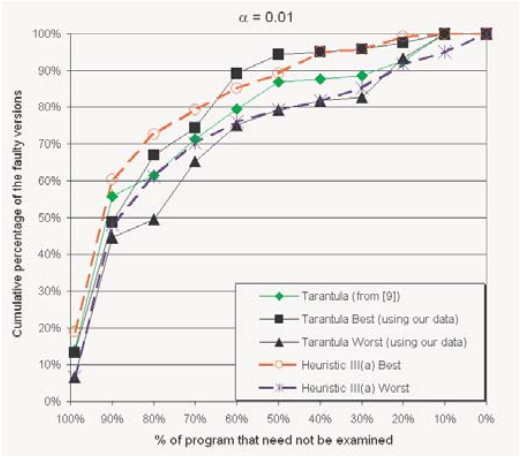

Figure 7. Comparison between Heuristic III (a) and Tarantula

Our next effort is to further improve the effectiveness of our method such that the best effectiveness of our method is better than the best effectiveness of Tarantula, and the worst effectiveness of our method is better than the worst effectiveness of Tarantula. In Heuristics III (b), (c), (d), and (e) different values are assigned to α. We observe that when α is decreased from 0.01 (Figure 7) to 0.0075 (Figure 8) to 0.0050 (Figure 9) to 0.0025 (Figure 10) and to 0.001(Figure 11), there is a continuous improvement of effectiveness from Heuristic III (a) to III (b) to III (c) to III (d) and to III (e). In particular for α = 0.0050, 0.0025, and 0.001, not only is the best effectiveness of Heuristic III better than the best effectiveness of Tarantula, but also the worst effectiveness of Heuristic III is better than the worst effectiveness of Tarantula. We also observe that decreasing α to a value smaller than 0.001 (e.g., 0.0001) gives little improvement in the effectiveness of our method. Due to space limits, the corresponding figures are not included. Based on these observations, it seems that with respect to the programs in the Siemens suite the "best" α in Equation (3) is around 0.001. We would like to point out that the number "0.001" is an empirical value and may change for different sets of programs and test cases. Refer to Section 4 for more discussion.

Heuristic III (b) versus Tarantula with α = 0.0075

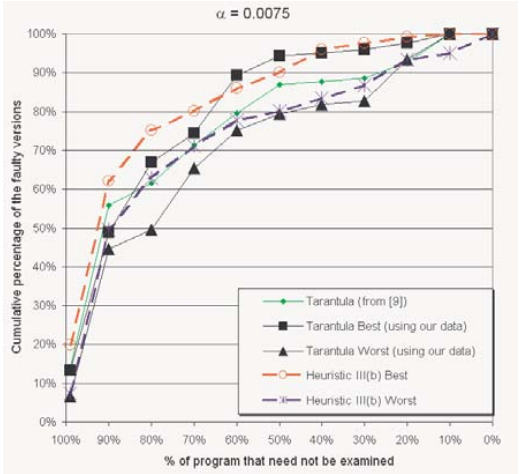

Figure 8. Comparison between Heuristic III (b) and Tarantula

Heuristic III (c) versus Tarantula with α = 0.0050

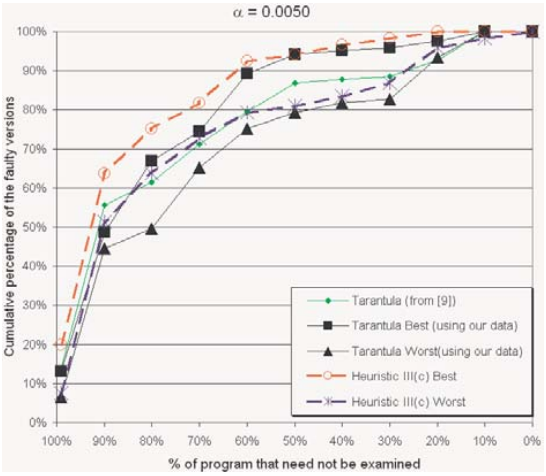

Figure 9. Comparison between Heuristic III (c) and Tarantula

Heuristic III (d) versus Tarantula with α = 0.0025

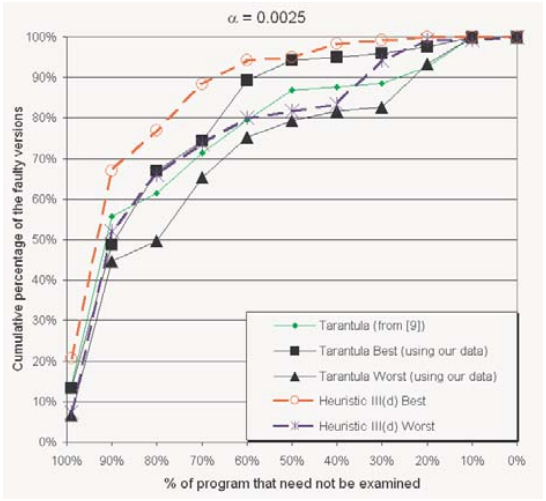

Figure 10. Comparison between Heuristic III (d) and Tarantula

Heuristic III (e) versus Tarantula with α = 0.001

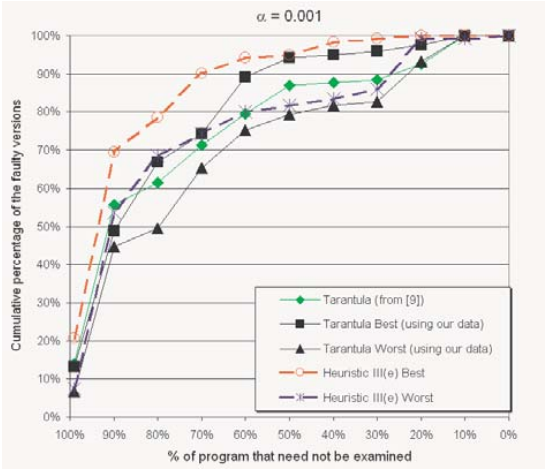

Figure 11. Comparison between Heuristic III (e) and Tarantula

0-7695-2870-8/07 $25.00

4. Discussion

Below we discuss a few issues related to the method we presented in Section 2.

Impact of α in Equation (3) on the effectiveness of fault localization

Different programs in our study have different numbers of test cases. This number varies from 1052 to 5542 of which most are successful tests. Let $\chi_{S/F}$ be the ratio of the number of successful tests that execute a faulty statement to the number of failed tests that execute the same faulty statement. Table 2 gives the cumulative percentage of the faulty programs that have some faulty statements whose $\chi_{S/F}$ is smaller than a certain value. For example, 84.30% of the faulty programs have at least one faulty statement whose $\chi_{S/F}$ is smaller than 100. Our intuition suggests that although the contribution of each additional successful test after the first few successful tests (e.g., the first two in our case) should not be completely ignored, they should not be overweighted either. With this understanding, the value of α in Equation (3) should be at most 0.01 or even smaller in order to guarantee that for at least one faulty statement in the corresponding 84.30% of the faulty programs, the contribution of its successful tests does not outweigh the contribution of its failed tests. Recall that α is the ratio of the contribution of each additional successful test to the contribution of each additional failed test after the 11th successful test.

Similarly, from Table 2 we observe that 87.60%, 91.74%, 97.52%, and 98.35% of the faulty programs have at least one faulty statement whose $\chi_{S/F}$ is smaller than 133.33, 200, 400, and 1000, respectively. Hence, when α is decreased to 0.0075 to 0.0050 to 0.0025 and to 0.001, more faulty programs (from 87.60% to 98.35%) have at least one faulty statement with the contribution of its successful tests adjusted not to outweigh the contribution of its failed tests. This is why there is a continuous improvement of the effectiveness of our method from Heuristic III (a) to III (b) to III (c) to III (d) and to III (e).

We also observe that when $\chi_{S/F}$ is increased from 1000 to 10000, the cumulative percentage of the faulty programs is only increased from 98.35% to 100.00%. Hence, when α is decreased from 0.001 to 0.0001, the improvement of the effectiveness is less obvious.

	Cumulative % of faulty programs		Cumulative % of faulty programs
$\chi_{S/F} < 100$	84.30	$\chi_{S/F} < 400$	97.52
$\chi_{S/F} < 133.33$	87.60	$\chi_{S/F} < 1{,}000$	98.35
$\chi_{S/F} < 200$	91.74	$\chi_{S/F} < 10{,}000$	100.00

Table 2. Cumulative % of faulty programs and their $\chi_{S/F}$ values

Bugs due to missing code

One may argue that the method described in Section 2 cannot find the location of a bug that is introduced by missing code. The reason is that if the code that is responsible for the bug is not even in the program, it cannot be located in any statement. Thus, there is no way to locate such a bug by using the proposed method. Although this might seem to be a good argument, it does not tell the whole story. We agree that missing code cannot be found in any statement. However, such missing code might, for example, have an adverse impact on a decision and cause a branch to be executed even though it should not be. If that is the case, the abnormal execution path with respect to a given test can certainly help us realize that some code required for making the right decision is missing. A programmer should be aware that the unexpected execution of some statements by certain tests can indicate the omission of certain important code which would affect control flow. Another example is that by examining why certain code is highlighted in high risk, we may also realize that some code in its neighborhood is missing.

Bugs due to a sequence of test cases

Sometimes a program fails not because of the current test but because of a previous test which, for example, fails to set up an appropriate execution environment for the current test. Let us assume that a test case t_α by itself does not reveal any bug, but it needs to be executed first in order to initiate a process before another test t_β can be executed correctly. If this process is not initiated properly by t_α, the program execution on t_β will fail. A good solution is to bundle these two test cases (t_α and t_β) together as one single failed test.

5. Related Studies

In this paper we focus on the comparison between the effectiveness of our method in fault localization with that of a series of other studies using the Tarantula, set-union, set intersection, nearest-neighbor, and cause-transitions techniques. Such a comparison is possible because all the studies use the same set of programs from the Siemens suite.

Jones and Harrold, et al., [8,9] present a fault-localization approach that ranks all the statements in a target program by their likelihood of containing defects. The approach utilizes an experimental formula to compute the suspiciousness of each statement based on the corresponding statement coverage information collected in executions of multiple tests, including both successful and failed tests. After all the statements in the target program are ranked, developers can examine them one by one until the defect is found. A tool, Tarantula, is created.

Renieris and Reiss [14] propose a nearest neighbor debugging approach that contrasts a failed test with another successful test which is most similar to the failed one in terms of the "distance" between them. In this approach, the execution of a test is represented as sequence of basic blocks that are sorted by their execution times. If a defect is in the difference set, it is located. For a defect that is not contained in the difference set, the approach can continue the defect localization by first constructing a program dependence graph and then including and checking adjacent un-checked nodes in the graph step by step until all the nodes in the graph are examined.

Two techniques based on the program spectra in [14] are the set union technique and the set intersection technique. The set union computes the set difference between the "program spectra" of a failed test and the union spectra of a set of successful tests. It focuses on the source code that is executed by the failed test but not by any of the successful tests. Such code is more suspicious than others. The set intersection technique excludes the code that is executed by all the successful tests but not by the failed test.

Cleve and Zeller [5] report a program state-based debugging approach, cause transition, to identify the locations and times where a cause of failure changes from one variable to another. This approach is based on their previous research on delta debugging [21,22]. An algorithm named *cts* is

proposed to quickly locate cause transitions in a program execution. A potential problem of the cause transition approach is that the cost of such an approach is relatively high; there may exist thousands of states in a program execution, and delta debugging at each matching point requires additional test runs to narrow down the causes.

We realize that there exist many other studies in program debugging such as Liblit, et al. [11], Liu, et al. [13] and slicing (static slicing [12,16], dynamic slicing [1,2,10], and execution slicing [4,18,19])-based techniques. In future work, we plan to design an experiment to compare these debugging techniques with those reported here in terms of their effectiveness in locating program bugs.

6. Conclusion & Future Work

We propose a code coverage-based fault localization method to help programmers effectively locate program bugs. We also propose that the contribution of all the successful test cases to program debugging should not be treated the same. We use a function to adjust the contribution of each successful test in computing the risk of code it executes. Data collected from our case study support our proposal.

We also examine the impact of α in Equation (3) (the ratio of the contribution of each additional successful test to the contribution of each additional failed test after the 11^{th} successful test) on the effectiveness of fault localization. When α is decreased from a small number such as 0.01 to 0.001, we observe a significant improvement of the effectiveness of our method, but such improvement becomes less obvious when α changes from 0.001 to 0.0001. For the programs in the Siemens suite, it seems that the "best" α is around 0.001.

Results from our study suggest that Heuristics III (c), (d) and (e) can perform better than other debugging techniques (including set-union, set intersection, nearest-neighbor, cause-transitions, and Tarantula) in locating program bugs, as they lead to the examination of a smaller percentage of the code than other techniques before the bugs are located.

In future work, we would like to conduct additional studies to compare the fault localization effectiveness of our method with other techniques using programs from different application domains. We would also like to further investigate the impact of α on the effectiveness of fault localization.

Reference

1. H. Agrawal, R. A. DeMillo, and E. H. Spafford, "Debugging with Dynamic Slicing and Backtracking," *Software – Practice & Experience*, 23(6):589-616, June, 1996.
2. H. Agrawal and J. R. Horgan, "Dynamic program slicing," *in Proceedings of the ACM SIGPLAN'90 Conference on Programming Language Design and Implementation*, June 1990.
3. H. Agrawal, J. R. Horgan, W. E. Wong, et al., "Mining System Tests to Aid Software Maintenance," *IEEE Computer*, 31(7):64-73, July 1998.
4. H. Agrawal, J. R. Horgan, S. London, and W. E. Wong, "Fault localization using execution slices and dataflow tests," *in Proceedings of the 6th IEEE International Symposium on Software Reliability Engineering*, pp. 143-151, Toulouse, France, October 1995.
5. H. Cleve and A. Zeller, "Locating causes of program failures," in *Proceedings of the 27th International Conference on Software Engineering (ICSE 2005)*, pp. 342-351, St. Louis, Missouri, USA, May 2005.
6. J. Ellsberger, D. Hogrefe, and A. Sarma, "SDL: Formal object-oriented language for communicating systems," Prentice Hall, 1997.
7. M. Hutchins, H. Foster, T. Goradia, and T. Ostrand, "Experiments on the effectiveness of dataflow- and controlflow-based test adequacy criteria," in *Proceedings of the 16th International Conference on Software Engineering*, pp. 191-200, Sorrento, Italy, May 1994.
8. J. A. Jones, "Fault localization using visualization of test information," in *Proceedings of 26th International Conference on Software Engineering (ICSE 2004)*, pp. 54-56, Edinburgh, Scotland, UK, May 2004.
9. J. A. Jones and M. J. Harrold, "Empirical evaluation of the tarantula automatic fault-localization technique," in *Proceedings of the 20th IEEE/ACM International Conference on Automated Software Engineering (ASE 2005)*, pp. 273-282, Long Beach, CA, USA, November 2005.
10. B. Korel and J. Laski, "Dynamic program slicing," *Information Processing Letters*, 29(3):155-163, October 1988.
11. B. Liblit, M. Naik, A. X. Zheng, A. Aiken, and M. I. Jordan, "Scalable Statistical Bug Isolation," in *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pp. 15-26, Chicago, IL, June 2005.
12. J. R. Lyle and M. Weiser, "Automatic program bug location by program slicing," in Proceedings of the 2nd International Conference on Computer and Applications, pp. 877-883, Beijing, China, June 1987.
13. C. Liu, L. Fei, X. Yan, J. Han, and S. P. Midkiff, "Statistical Debugging: A Hypothesis Testing-Based Approach," *IEEE Transactions on Software Engineering*, 32(10):831-848, October 2006.
14. M. Renieris and S. P. Reiss, "Fault localization with nearest neighbor queries," in *Proceedings of the 18th IEEE International Conference on Automated Software Engineering*, pp. 30-39, Montreal, Canada, October 2003.
15. The Siemens Suite, http://www-static.cc.gatech.edu/ aristotle/Tools/ subjects/, Januaury 2007.
16. M. Weiser, "Programmers use slices when debugging," *Communications of the ACM*, 25(7):446-452, July 1982.
17. W. E. Wong and J. J. Li, "An Integrated Solution for Testing and Analyzing Java Applications in an Industrial Setting," in *Proceedings of The 12th IEEE Asia-Pacific Software Engineering Conference* (APSEC), pp. 576-583, Taipei, Taiwan, December 2005.
18. W. E. Wong and Y. Qi, "Effective Program Debugging based on Execution Slices and Inter-Block Data Dependency," *Journal of Systems and Software*, 79(7):891-903, July 2006.
19. W. E. Wong, T. Sugeta, Y. Qi, and J. C. Maldonado, "Smart debugging software architectural design in SDL," *Journal of Systems and Software*, 76(1):15-28, April 2005.
20. χSuds User's manual, Telcordia Technologies, 1998.
21. A. Zeller, "Isolating cause-effect chains from computer programs," in *Proceedings of the 10th ACM SIGSOFT Symposium on Foundations of Software Engineering*, pp. 1-10, Charleston, South Carolina, USA, November 2002.
22. A. Zeller and R. Hildebrandt, "Simplifying and isolating failure-inducing input," *IEEE Transaction on Software Engineering*, 28(2):183-200, February 2002.