Hao D, Zhang L, Xie T *et al.* Interactive fault localization using test information. JOURNAL OF COMPUTER SCIENCE AND TECHNOLOGY 24(5): 962–974 Sept. 2009

Interactive Fault Localization Using Test Information

Dan Hao[1] (郝　丹), *Member*, *CCF*, *ACM*, Lu Zhang[1] (张　路), *Senior Member*, *CCF*, *Member*, *ACM*
Tao Xie[2] (谢　涛), Hong Mei[1,*] (梅　宏), *Senior Member*, *CCF*, and Jia-Su Sun[1] (孙家骕), *Senior Member*, *CCF*

[1] *Key Laboratory of High Confidence Software Technologies, Ministry of Education, Institute of Software School of Electronics Engineering and Computer Science, Peking University, Beijing 100871, China*

[2] *Department of Computer Science, North Carolina State University, Raleigh, NC 27695, U.S.A.*

E-mail: {haod, zhanglu, meih, sjs}@sei.pku.edu.cn; xie@csc.ncsu.edu

Received July 4, 2007; revised May 20, 2009.

Abstract Debugging is a time-consuming task in software development. Although various automated approaches have been proposed, they are not effective enough. On the other hand, in manual debugging, developers have difficulty in choosing breakpoints. To address these problems and help developers locate faults effectively, we propose an interactive fault-localization framework, combining the benefits of automated approaches and manual debugging. Before the fault is found, this framework continuously recommends checking points based on statements' suspicions, which are calculated according to the execution information of test cases and the feedback information from the developer at earlier checking points. Then we propose a naive approach, which is an initial implementation of this framework. However, with this naive approach or manual debugging, developers' wrong estimation of whether the faulty statement is executed before the checking point (breakpoint) may make the debugging process fail. So we propose another robust approach based on this framework, handling cases where developers make mistakes during the fault-localization process. We performed two experimental studies and the results show that the two interactive approaches are quite effective compared with existing fault-localization approaches. Moreover, the robust approach can help developers find faults when they make wrong estimation at some checking points.

Keywords debugging, fault localization, interactive approach

1 Introduction

Testing and debugging are essential parts of software development. Developers usually spend most of their development time on testing and debugging. A typical process of debugging consists of two steps[1]: finding the fault's location and fixing the fault. Most of existing researches on debugging focus on the former step, which is the focus of our research in this paper as well.

To reduce human effort in debugging, various automatic approaches[1−7] have been proposed to help locate faults. Among them, testing-based fault-localization (TBFL) approaches attract increasingly more attention in recent years. TBFL approaches use execution information to measure the suspicions of statements and rank the statements based on their suspicions. In the literature, researchers defined and used various terms to represent the measurement of suspicions. We unify these terms as "suspicion" in this paper. Intuitively, the suspicion of a statement comes from the statement's involvement in the failed test cases, because a failed test case must have executed a faulty statement. However, a passed test case may either have executed a faulty statement or not. For each TBFL approach, the higher suspicion a statement has, the earlier it appears in the ranked list. The developer can scan through the ranked list, checking whether the statements in the list contain faults. The benefits of these TBFL approaches are two-fold. First, these approaches help developers narrow the search space for the faulty statement. Thus developers can focus their attention only on suspicious statements. Second, statements are ranked by their suspicions, which help developers schedule statements for examination.

However, the effectiveness of the existing fault-localization approaches makes them difficult to serve

Regular Paper

*Corresponding Author

A preliminary version of this paper appeared in "Hao D, Zhang L, Mei H, Sun J. Towards interactive fault localization using test information. In *Proc. 13th Asia Pacific Software Engineering Conference*, 2006, pp.277–284."

This work is supported by the National Basic Research Program of China under Grant No. 2009CB320703, the National High-Tech Research and Development 863 Program of China under Grant No. 2007AA010301, the Science Fund for Creative Research Groups of China under Grant No. 60821003, the National Natural Science Foundation of China under Grant No. 60803012, and the China Postdoctoral Science Foundation Project under Grant No. 20080440254.

as a substitute for manual fault localization. The experimental results of these approaches[3−6] show that a developer still needs to examine about 10% or more of the total statements to find many faults. For a middle-sized program containing several thousand lines of statements, there may be many faults and developers may have to examine several hundred lines of statements to find each one of them. Moreover, if a developer directly applies these approaches to the debugging process, he has to look through the ranked list; this inspection process is quite different from the manual debugging habit.

Typically a developer uses some debugging tool provided by the development environment to help find faults in a program. When the developer cannot run the program successfully, he focuses his attention on the statements involved in this failed execution trace; these statements are suspicious because they have the probability of containing faults. The developer makes some hypothesis on the faulty program and sets one breakpoint. With the help of some debugging tool, the developer can access some variables' values at the breakpoint. With the help of the variables' values, the developer examines whether the location of the fault is at or very close to the breakpoint and/or determines whether the fault has been executed before the breakpoint. If the developer cannot find the fault in the examined statements, he or she will try to reduce the scope of suspicious statements. Typically, if the developer thinks that the faulty statement has been executed before a breakpoint, the scope of suspicious statements will be reduced to those statements executed before the breakpoint. Otherwise, the scope will be reduced to the statements that will be executed after the breakpoint. Then the developer repeats the preceding process until he or she finds the fault.

This manual debugging process is basically a binary search, which can be very effective, but its success and effectiveness depend on the developer's experience and debugging techniques. If the developer is not an experienced debugger, he would have to examine many statements before finding the fault or would even miss the faulty statement. In the following, we discuss two factors related to this situation. The first one is the choice of breakpoints. If the developer is not effective in setting breakpoints, the effectiveness of the manual debugging process might not be acceptable. To help the developer choose and set breakpoints, we propose an interactive fault-localization framework following the manual debugging routine, whose naive implementation is our naive interactive fault-localization approach. In this framework, statements' suspicions are calculated by the algorithm of an automatic fault-localization approach and the framework recommends a checking point① to the developer based on the statements' suspicions. Then the framework asks the developer to examine whether the statement at the checking point is faulty, and if not, determine whether the faulty statement is executed before the checking point. Based on the feedback information, the interactive framework modifies the suspicions and recommends another checking point.

The second factor that may impact the effectiveness of the manual debugging process is that the developer may have wrong estimation at some breakpoints when reducing the scope of suspicious statements. As the developer is possibly not quite familiar with the faulty program, he may determine a wrong program state as a correct one or vice versa. Furthermore, even if the developer is familiar with the program, he might not notice the incorrect program state if the developer does not examine the key variables related to the incorrectness. Due to this kind of mistakes, the scope of suspicious statements may be reduced to that of actually innocent ones. In such a circumstance, the developer would not find the faulty statement and the manual debugging process would fail. Even with the help of the naive approach based on the interactive localization framework, the developer might not find the fault if he or she makes such a mistake. The reason is that after the developer reduces the scope of suspicious statements, the approach reassigns the suspicions of statements outside the scope as 0. Thus, these statements will not be considered suspicious again because the naive approach does not ask the developer to examine any statement whose suspicion is 0.

To help the developer find the fault even in the case that he makes mistakes in reducing the scope of suspicious statements, we propose a robust approach based on the interactive fault-localization framework. The robustness of this approach assures that the statement containing faults will never be excluded and be found by the developer. The basic idea is to achieve robustness with a different strategy on suspicion modification in our interactive localization framework. When modifying the suspicions of the statements, the robust approach guarantees that a statement's suspicion would never be reduced to 0 unless the developer examines the statement at a checking point and finds it faultless. Therefore, the developer would always examine the faulty statement even if he makes mistakes at some checking points.

---

①A checking point has a similar meaning to a breakpoint. As a checking point is actually recommended to the developer for consideration as a breakpoint, we try to show the difference by using another term.

To evaluate our interactive framework including the naive interactive fault-localization approach and the robust interactive fault-localization approach, we conducted an experimental study on the Siemens programs and the Space program. This experimental study evaluates the robustness of the robust interactive approach. Moreover, by comparing with some existing fault-localization approaches, we got the following conclusions: 1) the naive interactive approach is usually more effective than all the other fault-localization approaches; 2) our robust approach is competitive in its fault-localization effectiveness compared with the automatic approaches.

As programmers may make wrong estimation on whether the faulty statement is executed before the checking point, we performed another experiment on the influence of the programmers' error estimation and the parameter α of our approach that determines to what extent the statements' suspicions are modified based on programmers' estimation. The results of this experiment provide support for the choice of α in the first experiment.

This paper makes the following contributions.

• We propose an interactive fault-localization framework, combining the characteristics of automated fault-localization approaches and manual debugging.

• Based on this interactive framework, we propose two concrete approaches including the naive approach as well as the robust approach. Specifically, the robust approach is proposed to handle the cases when developers make wrong estimations at some checking points.

• We performed two experimental studies, and the results demonstrate the effectiveness of our interactive approaches and the robustness of the robust approach.

The rest of the paper is organized as follows. Section 2 describes the details of the interactive framework and two proposed interactive approaches. Section 3 presents two experimental studies. Section 4 discusses related work on fault localization and Section 5 concludes the paper.

2 Approach

In this section we first give our interactive fault-localization framework, and then introduce the naive approach and the robust approach based on this framework. Finally, we discuss some issues in our approaches.

2.1 Interactive Fault-Localization Framework

The interactive fault-localization framework comprises three steps:

1) The framework gathers the execution information including which statements are executed in each test case and whether each test case is passed or failed.

2) The framework (re)calculates the suspicions of statements and provides a strategy on how to choose checking points.

3) The framework recommends a checking point and gathers human debugging information. In this process, the framework requires the developer to examine the checking point, and gathers the feedback information from the developer to modify suspicions.

The framework repeats the last two steps until the developer finds the fault. Fig.1 shows a more detailed overview of this framework.

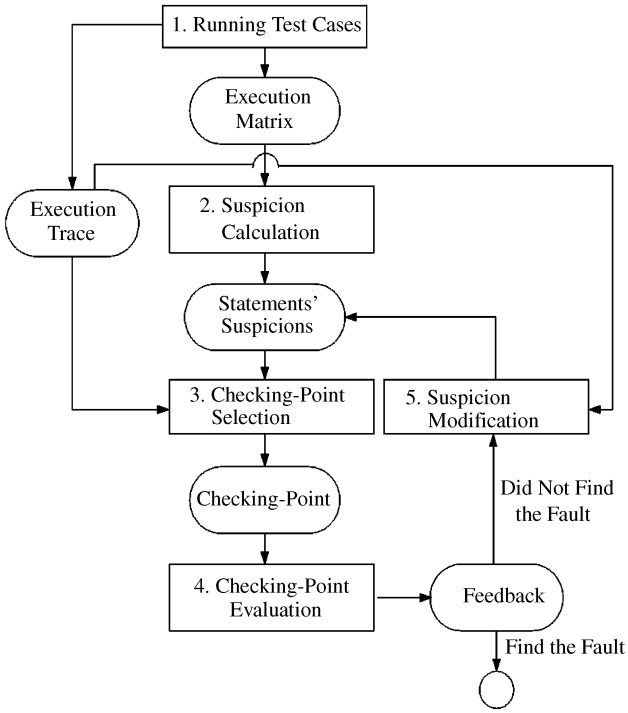

Fig.1. Interactive fault-localization framework.

First, the framework takes a test suite as input to run the faulty statement, gathering the execution information including the statements that are executed in each test case and whether each test case is passed or failed. Given a faulty program P consisting of statements $s_1, s_2, \ldots, s_m$ and a test suite $T = \{t_1, t_2, \ldots, t_n\}$, we record the execution information by an $n \times (m+1)$ Boolean matrix $\boldsymbol{E} = (e_{ij})$ $(1 \leqslant i \leqslant n$ and $1 \leqslant j \leqslant m+1)$,

$$e_{ij} = \begin{cases} 1, & \text{if } s_j \text{ is executed by } t_i \ (1 \leqslant j \leqslant m), \\ 1, & \text{if } t_i \text{ is passed } (j = m+1), \\ 0, & \text{otherwise,} \end{cases} \quad (1)$$

where $1 \leqslant i \leqslant n$. In this matrix, if statement s_j is executed by test case t_i, the corresponding element of the matrix is marked as 1. The last column of this Boolean matrix represents whether the test case is

passed or failed. If the output of a test case is consistent with expectation, the test case is considered passed, and the corresponding element in the matrix will be marked as 1; otherwise, the test case is considered failed, and the corresponding element in the matrix will be marked as 0.

The framework then asks the developer to choose a failed test case from the given test suite. The following fault-localization process is to locate the fault that this failed test case has revealed. The framework runs the faulty program again with this chosen failed test case, and gathers its execution trace. The reason is that the execution information gathered for the preceding matrix only includes whether a statement is executed in a test case, ignoring how many times this statement has been executed and the order of the statements in the execution. Here, an execution trace is a list of executed statements. The order of the statements in the list represents the order of execution. When a statement is executed more than once, the execution trace records all the executions of this statement in different positions of the list. If the faulty statement is executed more than once, the framework takes the first appearance② of the faulty statement as the location of the fault.

Second, our framework assigns initial suspicions to statements according to the algorithm on suspicion calculation of an automatic fault-localization approach, such as the Dicing approach[2], TARANTULA[1,10], and SAFL[4]. For the ease of implementation, our framework uses the algorithm of SAFL[4]. SAFL is a similarity-aware fault-localization approach, which uses the fuzzy set to represent each test case and thus eliminates the biased influence of similarity between test cases of locating faults. In general, the more failed test cases one statement is involved in, the more suspicious the statement is. Following this intuition, SAFL calculates the suspicion of statement s_j by the probability of the event that test cases executing statement s_j are failed. For statement s_j in program P, its initial suspicion, denoted as $P(j)$, is calculated with the following formula. Please refer to our previous work[4] to find more details.

$$P(j) = \frac{\sum_{k=1}^{m} \max\{f_{ik} | e_{ij} > 0 \wedge e_{i(m+1)} = 0 \wedge 1 \leqslant i \leqslant n\}}{\sum_{k=1}^{m} \max\{f_{ik} | e_{ij} > 0 \wedge 1 \leqslant i \leqslant n\}},$$

where

$$f_{ij} = \begin{cases} 1 \Big/ \sum_{k=1}^{m} e_{ik}, & \text{if } e_{ij} = 1, \\ 0, & \text{otherwise,} \end{cases} \quad (1 \leqslant i \leqslant n, 1 \leqslant j \leqslant m).$$

For the statements that are not executed by the chosen failed test case, the framework reassigns their suspicions as 0, because these statements have no contribution to the failure of this chosen test case. Given a faulty program whose lines of code are m and an input test suite that has n test cases, the time complexity of initial suspicion calculation is $O(m \times n)$.

Third, based on the statements' initial suspicions, the framework selects a checking point in the execution trace of the chosen failed test case according to the highest-value strategy, which comprises the following three rules.

- The basic principle is to choose the statement with the highest suspicion.
- When more than one statement has the same highest suspicion, choose the one that is executed the earliest of these statements in the execution trace.
- When the chosen statement is executed more than once by the selected failed test case, choose the first appearance of this statement in the execution trace as the checking point.

Fourth, the framework asks the developer to examine the chosen statement at the checking point to determine whether this statement contains the fault. If yes, the fault-localization process of this failed test case finishes. Otherwise, the framework requires the developer to determine whether the faulty statement has been executed before the checking point in the execution trace with the help of some debugging tool such as GDB③. Usually these debugging tools help provide some state information such as variables' values. Note that the framework only specifies the tasks the developer should finish, but not how the developer is supposed to finish the tasks. As state information is usually also helpful for determining whether a statement contains the fault, the developer may finish the two tasks together with the help of state information.

Fifth, the framework modifies statements' suspicions according to the developer's estimation at the checking point. If the developer determines that the faulty statement has been executed before the checking point, the framework would pay more attention to the statements executed before the checking point and decreases the suspicions of other statements. If the developer determines that the faulty statement will be executed after the checking point, the framework does the opposite. When the suspicion of a statement is set to 0, it will not be considered suspicious again in the following rounds of iteration. Based on the modified suspicions, the framework selects another checking point and

②We hold this assumption to ease the description and evaluation of our approach. However, our approach itself does not have such restriction.

③http://sourceware.org/gdb/.

continues the last three steps until the developer finds the fault.

2.2 Naive Interactive Approach

The naive interactive fault-localization approach is the initial implementation of our interactive fault-localization framework. In the following, we present the details how the naive approach modifies suspicions in the fifth step of the interactive framework.

Here, we use P_s to denote the set of statements whose suspicions are over 0. In the naive approach, P_s initially includes all the statements in the execution trace of the chosen failed test case. For a given checking point, the naive approach records the set of statements that are in P_s and have been executed before the checking point, excluding the chosen statement. Here, we denote this set as $S1$. Similarly, our approach also records the set of statements that are in P_s and will be executed after the checking point, excluding the chosen statement. We denote this set as $S2$. Note that $S1$ and $S2$ might have intersection because some statements that have been executed before the checking point might be executed again after the checking point.

After the developer examines the statement at the checking point and determines it to be faultless, the naive interactive approach modifies the suspicions of statements in P_s as follows. For statement s_j, whose current suspicion is $P(j)$, the naive approach defines its modified suspicion $P'(j)$ as follows.

• If the developer estimates that the faulty statement has been executed before the checking point according to the developer's estimation at the checking point, then

$$P'(j) = \begin{cases} P(j), & s_j \in S1, \\ 0, & s_j \in P_s - S1. \end{cases}$$

• If the developer estimates the faulty statement to be executed after the checking point, then

$$P'(j) = \begin{cases} P(j), & s_j \in S2, \\ 0, & s_j \in P_s - S2. \end{cases}$$

• If statement s_j is the chosen statement at the checking point, then $P'(j) = 0$.

2.3 Robust Interactive Approach

Also based on the interactive fault-localization framework, the newly proposed robust approach provides a new strategy to modify suspicions in the fifth step to achieve robustness. In the following, we present the details of this strategy.

Here, we still use P_s to denote the set of statements whose suspicions are over 0, $S1$ to denote the set of statements executed before the checking point, and $S2$ to denote the set of statements executed after the checking point. After the developer examines the statement at the checking point and determines it to be faultless, the robust interactive approach modifies the suspicions of statements in P_s as follows. For statement s_j, whose current suspicion is $P(j)$, the robust approach defines its modified suspicion $P''(j)$ follows.

• If the developer estimates that the faulty statement has been executed before the checking point according to the developer's estimation at the checking point, then

$$P''(j) = \begin{cases} P(j) \times (1+\alpha), & s_j \in S1, \\ P(j) \times (1-\alpha), & s_j \in P_s - S1. \end{cases}$$

• If the developer estimates the faulty statement to be executed after the checking point, then

$$P''(j) = \begin{cases} P(j) \times (1+\alpha), & s_j \in S2, \\ P(j) \times (1-\alpha), & s_j \in P_s - S2. \end{cases}$$

• If statement s_j is the chosen statement at the checking point, then $P''(j) = 0$.

Note that if the developer determines the statement at the checking point to be faultless, the suspicion of the statement is reassigned as 0. This means that the statement will never be considered suspicious again, as our suspicion-modification strategy can ensure that the suspicion of the statement will remain 0. In our suspicion-modification strategy, α can be a constant or variable, which determines the modification of suspicions. In our experiments, we evaluate the choice of α and its influence on the fault-localization effectiveness of the robust approach.

In fact, our robust approach can assure its robustness through the preceding suspicion-modification strategy, in which, the suspicion of any statement cannot be reassigned to 0 unless the developer examines it and determines to be faultless. Thus, if the developer makes a mistake at one checking point, this mistake will decrease the suspicion of the faulty statement. However, the correct estimations at the following checking points can probably increase the suspicion of the faulty statement, and bring it back to the attention of the developer.

2.4 Example

In this subsection, we use a program to show the robustness of our robust approach ($\alpha = 10\%$). The target program "Mid" is shown in Table 1, which is to find the median of three input data. There is one injected fault in this program, which is in line 7. The faulty statement is "$m = y$", which should be "$m = x$". The

given test suite includes four test cases t_1, t_2, t_3, and t_4. The input of t_1 is $\{3, 3, 5\}$, while the inputs of the other test cases are $\{2, 1, 3\}$, $\{5, 5, 3\}$, and $\{2, 3, 1\}$. We use • to represent which statements are executed in each test case, and the last row of the table gives whether the test case is passed or failed, where "P" represents passed, and "F" represents failed.

Table 1. Program Mid and Its Execution Trace

Mid() {	Test Suite			
int x, y, z, m;	t_1	t_2	t_3	t_4
	3, 3, 5	2, 1, 3	5, 5, 3	2, 3, 1
1. read(x, y, z);	•	•	•	•
2. $m = z$;	•	•	•	•
3. **if** $(y < z)$	•	•	•	•
4. **if** $(x < y)$	•	•		
5. $m = y$;				
6. **else if** $(x < z)$	•	•		
7. $m = y$;	•	•		
\\should be $m = x$;				
8. **else**			•	•
9. **if** $(x > y)$			•	•
10. $m = y$;				
11. **else if** $(x > z)$			•	•
12. $m = x$;			•	•
13. print(m);	•	•	•	•
}	P	F	P	P

First, the robust approach represents the execution information of Table 1 by a 4×14 Boolean matrix as

$$\begin{pmatrix} 1 & 1 & 1 & 1 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 1 & 1 \\ 1 & 1 & 1 & 1 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 1 & 1 & 1 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 1 & 1 & 1 & 1 \end{pmatrix}.$$

Then the robust approach assigns suspicions to these statements according to the description in Subsection 2.1, and the result is

s_j	1	2	3	4	5	6	7
$P(j)$	0.67	0.67	0.67	1	0	1	1
s_j	8	9	10	11	12	13	
$P(j)$	0	0	0	0	0	0.67	

lines 5, 8, 9, 10, 11, and 12 are not executed by any failed test case, so the robust approach assigns their suspicions as 0.

Suppose that the developer chooses failed test case t_2, whose input is 2, 1, and 3. The robust approach runs the program again with this test case and records its execution trace as $s_1^1, s_2^2, s_3^3, s_4^4, s_6^5, s_7^6, s_{13}^7$. Here s_j^i represents that statement s_j is executed by the chosen test case and that this statement is the i-th element of the execution trace. The following process is to locate the fault exposed by t_2. Then the robust approach selects the statement with the highest suspicion and recommends it to the developer as the checking point, which is line 4. Although lines 6 and 7 have the same suspicion as line 4, line 4 is selected because it is executed before lines 6 and 7. The developer checks line 4 and finds it correct. Then the robust approach asks the developer to determine whether the statements executed before the checking point contain a fault. If the developer correctly estimates that the faulty statement (which is in line 7) is executed after the checking point, the robust approach will recommend line 6 as the next checking point, followed by line 7. Thus, the developer can locate the fault after examining three checking points.

If the developer makes a mistake at the first checking point (line 4), the following fault-localization process will be as follows. Based on this (wrong) feedback information, the robust approach increases the suspicions of the statements executed before the checking point (including lines 1, 2, and 3), and decreases the suspicions of the statements executed after the checking point (including lines 6, 7 and 13). As the developer determines line 4 to be correct, the robust approach reassigns its suspicion as 0. Thus, the modified suspicions of these statements are

s_j	1	2	3	4	5	6	7
$P''(j)$	0.74	0.74	0.74	0	0	0.9	0.9
s_j	8	9	10	11	12	13	
$P''(j)$	0	0	0	0	0	0.60	

Then the robust approach recommends the next checking point, which is line 6. The developer thinks line 6 is correct. If the developer makes a mistake at this checking point again, then based on the wrong estimation the robust approach modifies the suspicions of these statements. The modified suspicions of lines 1, 2, 3 and 7 are all 0.81, which are higher than the suspicion of line 13 (which is 0.54).

The robust approach chooses line 1 as the next checking point. If the developer does not make wrong estimation any more, the following two checking points before finding the faulty statement would be line 2, and line 3, followed by line 7. Then the developer checks the statement in line 7 and finds it to be the faulty statement. From this process, we can see that the developer still finds the fault even if the developer makes mistakes at some checking points. The only difference is that the developer might need to examine more statements when there is some incorrect estimation. This example demonstrates the robustness of the robust approach.

However, in the manual debugging process or the debugging process with the naive approach, if the developer makes mistakes at a breakpoint (or checking

point), such as in line 4, the scope of suspicious statements will be reduced to lines 1, 2, and 3. The other statements of the faulty program will be taken as innocent. Then after examining these "suspicious" statements, the developer would not find the faulty statement and the fault-localization process fails.

2.5 Discussion

We next discuss some issues about our interactive approaches, including the issues related to the interactive framework and issues on our robustness.

2.5.1 Interactive Framework

First, our interactive framework is not specific to SAFL, although the framework uses the algorithm of SAFL to calculate the initial suspicions of statements. The framework can use various algorithms to calculate suspicions provided by different automatic fault-localization approaches[1−2]. In future work, we plan to empirically investigate the performance of our interactive approaches based on other fault-localization approaches.

Second, our current interactive framework does not provide any specific guidance for developers on how to choose an initial failed test case. However, a developer could select the test covering the fewest statements. Then the range of suspicious statements will be reduced initially. We plan to investigate this issue and propose techniques on failed test case selection in future work.

Third, our interactive framework chooses the statement with the highest suspicion as the checking point. This mechanism is advantageous because the developer can always examine the most suspicious statements, but this type of splitting does not assure to be beneficial for the subsequent round of interaction. In future work, we plan to develop a tradeoff strategy between splitting based on statements' suspicions and splitting the execution trace evenly.

2.5.2 Robustness

The robustness of our approach assures that a developer would not miss the faulty statement even if he makes wrong estimation at some checking point to determine whether the faulty statement has been examined before the checking point. Our robust approach achieves this robustness by its suspicion modification using $(1+\alpha)$ or $(1-\alpha)$, where α is currently taken as 10% in our robust approach. Intuitively, the suspicion modification is related to the developer's confidence on checking-point examination. We plan to further investigate other strategies depending on the developer's confidence.

However, in the debugging process, the developer may also make mistakes on examining whether the statement at the checking point contains the fault. The other fault-localization approaches make a similar assumption. These approaches assume that developers do not make mistakes while examining whether statements in the ranked list contain faults. Similar to these approaches[1−6], our approach currently assures that the developer does not make such mistakes. However, this robustness can be achieved by reducing the suspicion of a statement, but not to 0, when the developer thinks this statement is innocent.

3 Experimental Studies

To evaluate our approach, we performed two experimental studies.

The first is to evaluate the effectiveness of the proposed interactive framework compared with some existing automatic fault-localization approaches. Moreover, this experimental study is to evaluate the robustness of the interactive approach as well.

The second is to evaluate how the programmers' wrong estimation on whether the faulty statement is executed before the checking point and the parameter α influence the fault-localization results of the robust approach.

3.1 Subjects

We used the Siemens programs[8] and the Space program[9] as subjects in our experiments. These subject programs are all written in C. The overview of these subject programs is shown in Table 2. The second column of the table presents the functional descriptions of these programs. The third and fifth columns describe the number of faulty versions (abbreviated as "Ver") and the number of test cases. The fourth column includes two data: the data before the parentheses is the lines of code, abbreviated as LOC, whereas the data within the parentheses is the number of executable statements (abbreviated as Ex). Similar to the experimental setup of Jones and Harrold[10], our experiments used the executable statements instead of LOC. The number of executable statements is achieved by

Table 2. Overview of the Subject Programs

Program	Description	Ver	LOC (Ex)	Test
print_tokens	lexical analyzer	7	565(203)	4 072
print_tokens2	lexical analyzer	10	510(203)	4 057
replace	pattern replacer	32	563(289)	5 542
schedule	priority scheduler	9	412(162)	2 627
schedule2	priority scheduler	10	307(144)	2 683
tcas	altitude separator	41	173(67)	1 592
tot_info	info measurer	23	406(136)	1 026
Space	lang interpreter	38	9 564(6 218)	13 585

ignoring unexecutable source code such as blank lines, comments, macro definitions, function and variable declarations, and function prototypes. When a statement distributes in multiple lines, we take this statement as only one executable statement.

The Siemens programs[8,11] include seven programs, each of which has some test cases. For each program, there are several faulty versions, gathered from real experience. The Siemens program suite has 132 variants in total, but we only used 121 of them. Four versions were eliminated because the corresponding test suites did not reveal any fault. Five versions were eliminated because the instrumentation tool we used could not catch the execution information when the subject programs' execution incurs segmentation faults. Another two versions were eliminated because there are no syntactic differences between the correct versions and the faulty versions in their C files, but syntactic differences in H files. The instrumentation tool we used cannot catch the execution information of these H files. In the literature, the evaluation of existing fault-localization approaches typically eliminated some versions and used versions from 109 to 130 in the experiments[3,5−6,10].

The Space program is a larger program developed for the European Space Agency. The Space program has 38 faulty versions, each of which contains a single fault that was exposed during the development of the space program. We used only 21④ versions limited by the experimental cost. There are 13 585 test cases created by previous researchers[9,12]. There are 13 585 test cases for the Space program: 10 000 of them were randomly created by Vokolos and Frankl[9], and 3 585 of them were added by Rothermel *et al.*[12] to achieve better structural coverage.

3.2 Experimental Study 1

The first experimental study is to evaluate the effectiveness of our interactive fault-localization framework and the robustness of the proposed robust approach.

3.2.1 Process

First, we used GNU C compiler (i.e., gcc) to instrument and compile each faulty program. Then we ran the instrumented program with the given test suite and gathered the execution information by the "gcov" command. As the correct version for each subject program was available, we knew whether each test case was passed or failed by comparing the output of the faulty program and that of the correct version with the test case as input. Next we calculated the initial suspicions according to the technique in Subsection 2.1.

Second, we chose a failed test case that appeared the earliest in the input test suite. We instrumented the faulty program with a print statement after every executable statement and ran this instrumented program again with the chosen failed test case. It outputted an execution trace, which records the execution order of statements.

Third, we chose the checking point recommended by the highest-value strategy and examined the statement at the checking point. If we found that the statement at the checking point contained a fault⑤, then the faulty statement was found and the process of fault localization in this faulty program finished. Otherwise, we changed the statements' suspicions according to the developer's estimation at the checking point. Here, we use "correct estimation" to denote the situation that the developer thinks the faulty statement is executed before (or after) the checking point and the faulty statement is actually executed before (or after) the checking point. Otherwise, we denote it as "wrong estimation".

With the naive interactive approach, we assume that developers would always make correct estimation. Based on this assumption, our naive approach modified the suspicions of statements according to Subsection 2.2 and recommended another checking point. We repeated the preceding process until the developer finds the faulty statement. The number of recommended checking points during the preceding fault-localization process is taken as the result of the naive interactive approach.

However, the robust interactive approach is based on the fact that humans may make mistakes. As different developers may behave differently in the estimation, we simply assume that human behave in a totally random fashion and build the evaluation of the robust approach on top of this assumption. We define the concept of "accuracy rate" to represent the probability that the developer makes correct estimation at each checking point. As a result, for each checking point, we used the random function and the accuracy rate to determine whether the developer made correct estimation at a particular checking point. Specifically, we took the accuracy rate as 90% in this experimental study. That is, the developer makes correct estimation at 90% of the checking points. Moreover, when the accuracy rate is less than 100%, the developer's behavior at checking points is indeterminate due to the random function. Therefore, the corresponding results may become unstable. To deal with this issue, we repeated the fault-localization

④They are v3∼v5, v7, v8, v10∼v15, v17, v18, v20, v21, v23, v24, v28, v29, v31, and v33.

⑤A fault may distribute in multiple statements. In our experiments, if an approach finds any of them, the fault-localization process finishes.

process for each faulty program 50 times and used the average as the result of the robust approach with accuracy rate 90%.

3.2.2 Measurement

The compared automatic approaches output a ranked list of statements based on their suspicions, and then programmers examine these statements along the ranked list one by one until they find the faulty statement. Thus, in this experimental study, we used the number of statements, whose suspicions are no less than the suspicion of the faulty statement, as the cost of using these compared automatic approaches to locate the fault.

When applying the interactive approaches, programmers need to determine whether the faulty statement is executed before the checking point, besides determining whether the statement at the checking point is faulty. Thus, the cost of our interactive approach at each checking point includes two parts: 1) the cost of determining whether the statement at the checking point is faulty; and 2) the cost of estimating whether the faulty statement is executed before the checking point. For other compared automatic approaches, there is only cost of determining whether each listed statement is faulty. To make all the approaches comparable to each other, we transformed the second part of cost to the first part of cost for the interactive approaches. Specifically, we used "$1+s$" to denote the total cost of applying our interactive approaches to locate the faulty statement of some faulty program. Here, we use "1" to denote all the cost of determining whether the statements at checking-points are faulty, whereas "s" to denote all the cost of determining whether the faulty statement is executed before checking points. When "$s = 0$", the developer needs no extra cost of estimating whether the faulty statement is executed before checking points. That is to say, the efforts for determining whether the statements at checking points are faulty (such as checking the state information) can be directly used to determine whether the faulty statement has been executed before checking points. Similarly, when "$s = 1$", the cost of determining whether the statements at checking points are faulty is the same as the cost of estimating whether the faulty statement is executed before checking points. When "$s = 2$", the second part of cost is double the first part of cost.

Moreover, our measurement is based on the total cost of examination at checking points during the interactive fault localization process, not the cost of each checking point or statement. For example, during the fault localization process of a faulty program P, a programmer has set and made estimation at 30 checking points (which are $c_1, c_2, \ldots, c_{30}$) by using our interactive approach before finding the faulty statement. That is, during this process, this programmer examined whether the 30 statements at these checking points were faulty and whether the faulty statement had been executed before each of these checking points (except for the last checking point c_{30}). The statement at the last checking point is the faulty statement. The total cost on applying our interactive approach includes the cost of examining whether 30 statements are correct and the cost of determining whether the faulty statement has been executed before the 29 checking points. Our measurement views the relation between the former cost and the latter cost as "1" and "s", and the former cost is 30 statements, so the latter cost is estimated to be $30 \times s$ statements. Thus, the total cost of our interactive approach in this example is $30 \times (1+s)$ statements. Specifically, when "$s = 0$", the cost of our interactive approach is 30 (statements); when "$s = 1$", the cost of our interactive approach is 60 (statements); when "$s = 2$", the cost of our interactive approach is 90 (statements). Although the effort for estimating whether the statement at checking point c_i ($1 \leqslant c_i < 30$) is faulty may be useful for the estimation whether the faulty statement has been executed before c_i, it is tedious and impossible to list their relation at each checking point because various checking points tend to have different relations. To ease implementation, our measurement ignores the difference of these checking points and takes the cost of all the checking points during the fault localization process as a whole.

In practice, the relation between the cost of determining whether the statements at checking points are faulty and the cost of determining whether the faulty statement is executed before checking points is much more complex than "$1+s$". In the measurement of our experiments, we simplified their relation to ease and automate the comparison between interactive approaches and automatic approaches.

During the experiments, we applied our interactive approach to each target program and recorded the number of recommended checking points. Then we applied our measurement "$1+s$" ($s = 0$) to the number of recommended checking points and got the experimental results of our interactive approach when $s = 0$, applied "$1+s$" ($s = 1$) to the number of recommended checking points and got the results of our interactive approach when $s = 1$, applied "$1+s$" ($s = 2$) to the number of recommended checking points and got the results of our interactive approach when $s = 2$.

3.2.3 Results and Analysis

We summarize the experimental results of the seven programs of the Siemens programs and the Space

program in Table 3. In this table, we use "Robust Approach" to denote our proposed robust interactive approach, "Naive Approach" denote our naive interactive approach. Each item of Table 3 includes two data. The data before the parenthesis is the average results for each program, which stand for the number of statements that programmers need to examine before finding the first faulty statement. The data inside the parenthesis is the ratio between the average result (which is the data before the corresponding parenthesis) and the number of executable statements for each program. As discussed before, the cost of the interactive approach includes the cost of estimating whether the faulty statement is executed before the checking point and the cost of examining the statement at the checking point. So the cost for interactive approaches consists of these two costs, and we transfer the former cost to the latter cost. Table 3 gives the average results for the robust approach when $s = 0$, $s = 1$, and $s = 2$. However, for the naive approach, this table gives only the average results when $s = 0$ for simplicity.

According to Table 3, except for the results on print_tokens and schedule, the naive interactive approach achieves the best fault-localization results of all the fault-localization approaches. However, the naive approach requires that programmers provide *correct* estimation of each checking point. This assumption is far from real.

For our robust interactive approach, when $s = 0$, most of its results (except for print_tokens and schedule) are better than the other compared automatic fault-localization approaches. Even for print_tokens and schedule, although our robust approach with $s = 0$ is not as effective as the Dicing approach, the fault-localization results of our robust approach with $s = 0$ are close to those of the Dicing approach. That is, our robust interactive approach is as effective as the compared automatic fault-localization approaches, and is often more effective than them when $s = 0$. When $s = 1$ or $s = 2$, our robust approach shows its advantage over the other compared automatic approaches only on the Space program, but not on the small programs of the Siemens programs.

Based on the preceding results and analysis, we get the following conclusions for the subjects in this experimental study.

1) The naive interactive approach achieves almost all the best fault-localization results. However, its efficiency lies in its strict assumption.

2) The robust interactive approach is more effective than the other compared automatic approaches in most cases when $s = 0$.

3) The robust interactive approach is usually not as effective as the other approaches for the small programs in the Siemens programs when $s = 1$ or $s = 2$. However, the robust approach shows its efficiency for the largest program of our experimental studies (such as the Space program) when $s = 1$ or $s = 2$.

4) When the accuracy rate is 90%, which means that programmers sometimes make wrong estimation of whether the faulty statement is executed before the checking point, the proposed robust approach can still help the programmers find all the faulty statements. That is, the proposed robust approach is robust.

3.3 Experimental Study 2

In this experimental study, we will investigate how the parameter α in our robust approach and the accuracy rate influence the fault-localization results of our robust approach.

3.3.1 Process

In this experiment, the parameter α is set to 5%, 10%, 15%, 20%, and 50%, respectively, whereas the accuracy rate is taken as 60%, 70%, 80%, 90%, and 100%, respectively. For each given parameter α and accuracy rate, we repeat the process of applying our robust approach to some faulty versions of the Space program, recording how many checking points have been set before finding the faulty statement. Then we take this number as the result for each faulty Space program in the experimental study. For any given α and accuracy rate, we use the average results as the experimental results.

Table 3. Comparison of Experimental Rosults

	Robust Approach $(1 + s)$			Naive Approach	SAFL	Dicing	TARANTULA
	$s = 0$	$s = 1$	$s = 2$	$(s = 0)$			
print_tokens	10 (5.12%)	20 (9.75%)	29 (14.38%)	7 (3.55%)	38 (18.52%)	6 (3.05%)	15 (7.29%)
print_tokens2	15 (7.28%)	29 (14.07%)	42 (20.85%)	6 (2.96%)	17 (8.54%)	31 (15.33%)	33 (16.15%)
schedule	12 (7.41%)	23 (14.20%)	34 (20.99%)	13 (8.15%)	31 (19.01%)	11 (6.54%)	11 (6.54%)
schedule2	43 (29.51%)	84 (58.33%)	126 (87.15%)	33 (22.57%)	99 (68.66%)	103 (71.79%)	90 (62.59%)
tcas	10 (15.30%)	20 (29.10%)	29 (42.91%)	7 (10.07%)	37 (55.78%)	33 (48.58%)	31 (46.46%)
tot_info	14 (10.23%)	27 (19.73%)	40 (29.23%)	9 (6.59%)	34 (25.15%)	44 (32.54%)	33 (24.57%)
replace	20 (6.79%)	38 (13.24%)	57 (19.69%)	11 (3.93%)	62 (21.38%)	26 (8.89%)	28 (9.57%)
space	53 (0.86%)	105 (1.70%)	157 (2.53%)	13 (0.21%)	278 (4.47%)	303 (4.87%)	176 (2.83%)

3.3.2 Results and Analysis

The results for the second experiment are shown in Fig.2, which depicts the fault-localization results of different accuracy rates for any given α. The horizontal axis in Fig.2 is the accuracy rate, whereas the vertical axis is the average fault-localization results.

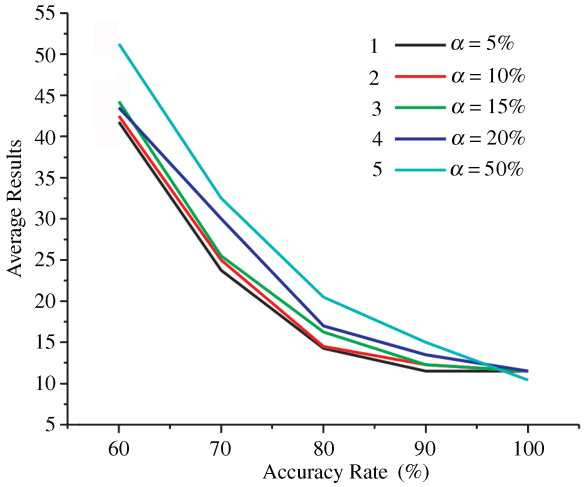

Fig.2. Results of different accuracy rates and α.

From this figure, for any given accuracy rate less than 95%, the curve for $\alpha = 5\%$ is below all the other curves, then follows $\alpha = 10\%$, 15%, 20%, and 50%. Therefore in this experiment, the fault-localization results of $\alpha = 5\%$ are better than those of $\alpha = 10\%$, then follows $\alpha = 15\%$, 20%, and 50%. That is, in this small experiment, the smallest value of parameter α (which is 5%) helps our robust approach achieve the best fault-localization results no matter what the value of the accuracy rate is. Although our robust approach achieves close results when $\alpha = 5\%$ and 10%, the results of our robust approach with $\alpha = 20\%$ or 50% are much worse than $\alpha = 5\%$ or $\alpha = 10\%$. Therefore, in the first experimental study, the choice of α, which is 10%, is acceptable.

3.4 Threats to Validity

The main threats to the external validity lie in the subject programs. The subject programs were all written in C, and the scale of the programs is not large enough to represent real programs. Additionally, each faulty version has only one fault. We plan to conduct more experiments on larger programs with multiple faults written in various languages, including C++ and Java, etc.

The main threat to the internal validity is the choice of the failed test case, which might be biassed towards the experimental results. For a faulty program, choosing a different failed test case might lead to different experimental results. To reduce this threat, we have applied our approaches on many different faulty versions of each subject program, and each different version implies a choice of a different failed test case. We plan to further reduce this threat through conducting experiments with randomly selected failed test cases.

The main threat to the construct validity lies in our simulation of the developer's behavior of making mistakes. Our experiments assume that the developer makes mistakes in a random fashion when estimating whether the fault is executed before the checking point. However, many factors could affect the developer's behavior, including the experience, the target faulty program and so on. So in our future work, we plan to simulate the developer's behavior based on some psychological models such as the Cognitive Walkthrough method[13] considering these factors.

4 Related Work

Algorithmic debugging[14] determines whether a procedure contains faults by the developers' answer on whether the procedure outputs as expected. Fritzson *et al.*[15] used the category partition testing method to reduce developers' effort on answering questions and combined the slicing technique to further narrow the suspicious scope. However, the algorithmic debugging is to locate the fault at the procedure level, not the statement level.

Besides the preceding interactive approaches, there are many automatic fault-localization approaches that primarily rely on test information. These approaches analyze the execution traces of a large number of test cases and provide a ranked list of statements. The original Dicing approach[2] is based on the concept of "dice", which is the set of statements in the execution slice of a failed test case subtracted by the statements in the execution slice of a passed one. However, the statements within a dice are equally suspicious to the programmers. Moreover, if the faulty statement is executed by both the failed test case and the passed test case, this faulty statement will be not in the dice of these two test cases, and the programmer will not find this faulty statement with this Dicing approach. To address these two problems, in our experiments, we extend the original Dicing approach by using many failed test cases and passed test cases, and calculating all the dices between each failed test case and each passed test case. The more dices one statement belongs to, the more suspicious this statement is. TARANTULA[1] defines a pair of a color component and a brightness component to rank the statements, which is based on the intuition that the more failed test cases a statement is involved in,

the more suspicious it is. Later, Jones and Harrold[10] transferred the formulae of the preceding two components into a more mathematical form. The Nearest Neighbor Queries approach[6] compares the spectra on the execution traces of a failed test case and a passed test case to produce a report of "suspicious" parts of the program. To eliminate the influence of uneven distribution of test cases on fault localization, SAFL[4] takes each test case as a fuzzy set and exploits the probability theory to calculate the suspicions of statements. CT[3] analyzes the dataflow between the variables, which are related to the failure of a failing run. Liblit *et al.*[16−17] instrument predicates in the source code to gather information from program runs and rank the predicates on their conditional probability to locate the fault. However, whether the instrumentation is executed or not is decided on a coin flip. Similarly, SOBER[5] locates the fault by comparing the predicates involved in both passed test cases and failed test cases. These two approaches are both based on the collection of predicates, whereas our interactive approaches are based on the execution information. Besides the preceding approaches, Zhang *et al.*[18] explored the effectiveness of three dynamic slicing techniques (data slicing, full slicing and relevant slicing) on fault localization. Their later work[19] proposed the concept of critical predicates, and uses critical predicates with other slicing techniques to locate faults. Different from these automated approaches, our approaches are interactive, which combine the test information and feedback information from the developer in the fault-localization process.

There are also some other approaches that automatically detect anomalies in execution traces and further use anomalies to locate faults. For example, some early approaches[20] search for unusual data flows that occur in each individual execution trace. These anomalies are used for assisting fault localization. The DIDUCE approach[21] uses the execution of passed test cases to infer invariants and violations of the invariants in failed test cases to infer the locations of faults.

5 Conclusion

Although various automated fault-localization approaches have been proposed to locate the fault in the faulty program, these approaches are not effective enough, and require the developer to inspect a list of suggested positions before finding the fault. Compared with these automated approaches, manual debugging can be effective for experienced developers. However, two major difficulties lie in manual debugging. The first one is the choice of breakpoints. The second one is that the developer's wrong estimation at breakpoints would lead to the failure of manual debugging. To solve the first difficulty, we proposed an interactive fault-localization framework, combining the advantages of automated fault localization and manual debugging. This framework calculates the suspicions of statements based on the execution information of test cases, and then recommends checking points according to the suspicions of statements. The naive interactive approach is just an initial implementation of this framework. To solve the second difficulty in manual debugging, we proposed a robust technique based on this framework, which achieves its robustness by its strategy on suspicions modification. We performed two experimental studies, and drew the following conclusions. First, the robust approach can help the developer locate the fault even if the developer randomly makes mistakes at some checking points. Second, our interactive approaches, including the naive approach and the robust approach, achieve promising effectiveness compared with existing fault-localization approaches.

References

[1] Jones J A, Harrold M J, Stasko J. Visualization of test information to assist fault localization. In *Proc. the 24th Int. Conf. Software Engineering*, Orlando, Florida, USA, May 19–25, 2002, pp.467–477.

[2] Agrawal H, Horgan J, London S, Wong W. Fault location using execution slices and dataflow tests. In *Proc. the 6th Int. Symp. Software Reliability Engineering*, Toulouse, France, Oct. 24–27, 1995, pp.143–151.

[3] Cleve H, Zeller A. Locating causes of program failures. In *Proc. the 27th Int. Conf. Software Engineering*, St. Louis, Missouri, USA, May 15–21, 2005, pp.342–351.

[4] Hao D, Pan Y, Zhang L, Mei H, Sun J. A similarity-aware approach to testing based fault localization. In *Proc. the 20th IEEE Int. Conf. Automated Software Engineering*, Long Beach, CA, USA, Nov. 7–11, 2005, pp.291–294.

[5] Liu C, Yan X, Fei L, Han J, Midkiff S P. SOBER: Statistical model-based bug localization. In *Proc. the 13th ACM SIGSOFT Symp. Foundations of Software Engineering*, Lisbon, Portugal, Sept. 5–9, 2005, pp.286–295.

[6] Renieris M, Reiss S P. Fault localization with nearest neighbor queries. In *Proc. the 18th Int. Conf. Automated Software Engineering*, Montreal, Canada, Oct. 6–10, 2003, pp.30–39.

[7] Misherghi G, Su Z. HDD: Hierarchical delta debugging. In *Proc. the 28th IEEE Int. Conf. Software Engineering*, Shanghai, China, May 20–28, 2006, pp.20–28.

[8] Hutchins M, Foster H, Goradia T, Ostrand T. Experiments on the effectiveness of dataflow- and control-flow-based test adequacy criteria. In *Proc. the 16th Int. Conf. Software Engineering*, Sorrento, Italy, May 16–21, 1994, pp.191–200.

[9] Vokolos F I, Frankl P G. Empirical evaluation of the textual differencing regression testing technique. In *Proc. the 14th Int. Conf. Software Maintenance*, Bethesda, Maryland, USA, Nov. 16–19, 1998, pp.44–53.

[10] Jones J A, Harrold M J. Empirical evaluation of tarantula automatic fault-localization technique. In *Proc. the 20th IEEE Int. Conf. Automated Software Engineering*, Long Beach, CA, USA, Nov. 7–11, 2005, pp.273–282.

[11] Rothermel G, Harrold M J. Experimental studies of a safe regression test selection technique. *IEEE Trans. Software Engineering*, 1998, 24(6): 401–419.

[12] Rothermel G, Untch R H, Chu C, Harrold M J. Prioritizing test cases for regression testing. *IEEE Trans. Software Engineering*, 2001, 27(10): 929–948.
[13] Wharton C, Rieman J, Lewis C, Polson P. The Cognitive Walkthrough Method: A Practitioner's Guide. John Wiley, 1994.
[14] Shapiro E Y. Algorithm Program Debugging. MIT Press, 1983.
[15] Fritzson P, Shahmehri N, Kamkar M, Gyimothy T. Generalized algorithmic debugging and testing. *ACM Letters on Programming Languages and Systems*, 1992, 1(4): 303–322.
[16] Liblit B, Naik M, Zheng A X, Aiken A, Jordan M I. Scalable statistical bug isolation. In *Proc. the ACM SIGPLAN Conf. Programming Language Design and Implementation*, Chicago, IL, USA, June 12–15, 2005, pp.15–26.
[17] Liblit B, Aiken A, Zheng A X, Jordan M I. Bug isolation via remote program sampling. In *Proc. the ACM SIGPLAN Conf. Programming Languages Design and Implementation*, San Diego, California, USA, June 9–11, 2003, pp.141–154.
[18] Zhang X, He H, Gupta N, Gupta R. Experimental evaluation of using dynamic slices for fault localization. In *Proc. the 6th Int. Symp. Automated and Analysis-Driven Debugging*, Monterey, California, USA, Sept. 19–21, 2005, pp.33–42.
[19] Zhang X, Gupta N, Gupta R. Locating faults through automated predicate switching. In *Proc. the 28th IEEE Int. Conf. Software Engineering*, Shanghai, China, May 20–28, 2006, pp.20–28.
[20] Demillo R, Pan H, Spafford E. Failure and fault analysis for software debugging. In *Proc. the 21st Int. Computer Software and Application Conf.*, Washington DC USA, Aug. 11–15, 1997, pp.515–521.
[21] Hangal S, Lam M S. Tracking down software bugs using automatic anomaly detection. In *Proc. the 24th Int. Conf. Software Engineering*, Orlando, Florida, USA, May 19–25, 2002, pp.291–301.

Dan Hao is a postdoctoral researcher in the School of Electronics Engineering and Computer Science at Peking University. She is a member of CCF and ACM. She received the Ph.D. degree from School of Electronics Engineering and Computer Science, Peking University in 2008. Her current research interests include software testing, debugging, and program comprehension.

Lu Zhang is an associate professor in the School of Electronics Engineering and Computer Science at Peking University. He is a member of ACM and a senior member of CCF. He received the B.S. and the Ph.D. degrees in computer science both from Peking University. He was a postdoctoral researcher with the Department of Computer Science, the University of Liverpool in April 2001∼January 2003. His current research interests include program comprehension, reverse engineering, component-based software development, software modeling, and software testing.

Tao Xie is an assistant professor in the Department of Computer Science of the College of Engineering at North Carolina State University. He received his Ph.D. degree in computer science from the University of Washington in 2005. Before that, he received an M.S. degree in computer science from the University of Washington in 2002 and an M.S. degree in computer science from Peking University in 2000. His research interests are in software engineering, with an emphasis on improving software dependability and productivity.

Hong Mei is a professor at the School of Electronics Engineering and Computer Science, Peking University and the director of Special Interest Group of Software Engineering of CCF. He received his Ph.D. degree in computer science from Shanghai Jiaotong University in 1992. His current research interests include software engineering and software engineering environment, software reuse and software component technology, distributed object technology, software production technology and programming language.

Jia-Su Sun is a professor at the School of Electronics Engineering and Computer Science at Peking University and a senior member of CCF. His research interests include programming languages, program comprehension, reverse engineering and reengineering. He has published more than 30 papers and led many State research projects.