State Dependency Probabilistic Model for Fault Localization

Gong Dandan*, Wang Tiantian, Su Xiaohong, Ma Peijun, Wang Yu

School of Computer Science and Technology, Harbin Institute of Technology, Harbin 150001, China

Abstract

Context: Fault localization is an important and expensive activity in software debugging. Previous studies indicated that statistically-based fault-localization techniques are effective in prioritizing the possible faulty statements with relatively low computational complexity, but prior works on statistical analysis have not fully investigated the behavior state information of each program element.

Objective: The objective of this paper is to propose an effective fault-localization approach based on the analysis of state dependency information between program elements.

Method: In this paper, state dependency is proposed to describe the control flow dependence between statements with particular states. A state dependency probabilistic model uses path profiles to analyze the state dependency information. Then, a fault-localization approach is proposed to locate faults by differentiating the state dependencies in passed and failed test cases.

Results: We evaluated the fault-localization effectiveness of our approach based on the experiments on Siemens programs and four UNIX programs. Furthermore, we compared our approach with current state-of-art fault-localization methods such as SOBER, Tarantula, and CP. The experimental results show that, our approach can locate more faults than the other methods in every range on Siemens programs, and the overall efficiency of our approach in the range of 10%-30% of analyzed source code is higher than the other methods on UNIX programs.

* Corresponding author. Tel: 86-451-86412824  Fax: 86-451-86412824
E-mail: gongdandan0418@126.com
Conclusion: Our studies show that our approach consistently outperforms the other evaluated techniques in terms of effectiveness in fault localization on Siemens programs. Moreover, our approach is highly effective in fault localization even when very few test cases are available.

**Keywords:** fault localization; statistical analysis; control flow graph

1. Introduction

Software debugging, which includes locating and correcting faulty program statements, is an important but tedious activity in software development and maintenance. Fault localization is the most expensive activity in debugging [1]. Traditionally, fault localization is a manual task which only depends on developers’ experience, and this manual process is tedious and error-prone. Thus, it is necessary to find an efficient and effective method to help programmers locate faults.

In recent years, many efforts have been made to help programmers locate faults, such as program slicing [2-9], state altering [10-13], model-based methods [14-18], and statistical analysis [19-28]. Among various fault-localization approaches, statistical analysis has proven to be effective in prioritizing the possible faulty statements with relatively low computational complexity [22]. It locates fault-relevant statements by comparing the statistical differences between program elements (e.g., statements, predicates, etc.) executed by passed and failed test cases. In terms of different analysis objects, there are three main types of statistically-based fault localization. The first method is traditional statistically-based fault localization. Typical examples of such techniques are CBI [20], SOBER [24,25] and Tarantula [19]. CBI captures predicates coverage information and computes suspiciousness for certain types of predicates. It can separate the effects of different bugs and identify predictors that are associated with individual bugs. It only analyzes whether the predicates have been executed or not in runtime, rather than distinguishes the number of times the predicates have been executed. In contrast, SOBER can calculate the number of times each predicate is evaluated to be true or false, respectively. It ranks the suspicious predicates based on the hypothesis that evaluations of the predicates in incorrect runs are significantly different
from those in correct runs. However, the above two methods can only locate the faults which are relevant to predicates. Tarantula can rank all the executable statements by their likelihood of containing faults. It is based on the assumption that a statement, which is primarily executed by failed test cases rather than by passed test cases, is more likely to be faulty. But this approach has several limitations. Firstly, it only considers whether a statement has been executed, but does not analyze the execution frequency of the statement. Secondly, paths are an effective way of isolating program failures because it can capture more information about program execution behavior [21]. However, Tarantula only considers the coverage information of each executable statement, which is not enough to analyze the program execution behavior. The necessary consequence would be reduced accuracy of fault localization. The second method analyzes the control dependence of program. This method captures more information about program execution behavior or the statistical dependences between program elements in a way that facilitates making probabilistic inferences about program behaviors. Typical example is CP [30]. CP uses edge profiles to represent program executions, and then model how each basic block contributes to failures by capturing the suspicious propagation of program states abstractly. Existing coverage-based fault-localization approaches focus on assessing the suspiciousness of individual program entities, but ignore the structural relationships among statements. This edge-oriented technique captures the relationships among statements based on the analysis of control dependence of program, so this method is more effective than coverage-based fault localization because the semantic of a program can be analyzed. The third method analyzes data dependence as well as control dependence. Typical examples are RankCP [32] and a method presented in [31]. Feng et al. [31] present universal probabilistic models to characterize the behaviors of various instruction types, and then construct a Bayesian network called the Error Flow Graph. Standard inference algorithms are employed to compute the probability of each executed statement being faulty. RankCP [32] presents probabilistic program dependence graph to facilitate probabilistic analysis and reasoning about uncertain program behavior. However, the analysis of data dependence is expensive [33-34].
To address the above limitations, we use path profiles to capture state dependence information about program executions rather than only use basic predicate profiles or program coverage information. A state dependency probabilistic model is defined to capture statistical information about program execution behavior. In this model, execution states of each program element are analyzed, state dependency is proposed to describe the control flow dependence between statements with particular states, and the probabilities of state dependencies are calculated according to execution paths and frequencies. The state dependency probabilistic model can indicate how a failing execution differs from successful ones, both structurally and statistically, so it can be an effective approximate model for representing behaviors of a program for fault localization.

If a state dependency is frequently executed in failed test cases, and rarely executed in passed test cases, this state dependency is more likely to be faulty. Based on this assumption, suspicious statements can be ranked and localized by analyzing the state dependency probabilistic model.

The main contributions of this paper include:

- We present a novel model (state dependency probabilistic model, SDPM) to capture state dependence information about the execution path and the execution behavior of a program.
- We propose a statistical model-based fault-localization approach to improve fault-localization effectiveness.
- The experiments are designed and implemented to verify the effect of our approach. Final results show that our approach is effective in locating faults.

The rest of the paper is organized as follows: Section 2 presents the state dependency probabilistic model. Section 3 describes the fault localization based on state dependency probabilistic model. Section 4 reports empirical studies we performed to evaluate our approach. Section 5 presents related work and the conclusion is given in Section 6.

2. State Dependency Probabilistic Model

A control flow graph (CFG) can clearly reveal all possible paths of program
execution. We use CFGs to analyze the control flow dependence between program elements. Fig. 1 shows the CFG of the example program mid(). The program mid() has been used in many previous fault-localization studies [19] and test-suite reduction approaches [35-37] as example program. This program inputs three integers and outputs the median value of the three integers. It contains a fault on line 3—this line should be “if (x<y)”.

**Definition 1 (Control Flow Graph, CFG):** The Control Flow Graph G=(N,E) for program P is a directed graph, which consists of a set of nodes N and a set of directed edges E, where N represents the program’s statements s₁, …, sₙ, and E is a set of edges which show how the program can move from statement to statement. Statement sᵢ is control flow dependent on sⱼ if there is an edge from statement sᵢ to statement sⱼ, and sᵢ is called a parent of sⱼ, sⱼ is a child of sᵢ. The parent node of statement sᵢ is denoted as para(sᵢ). If the statement sⱼ has multiple parent nodes, the k-th parent node of sⱼ is denoted as paraⱼ(sⱼ).

We now describe in detail the step of the SDPM generation process that is novel for our technique: the calculation method of state dependency probability for each program element.

The “execution state” represents the behavior state of node during program execution. For each statement, two states were specified, which represent whether the statement is executed or not. When the selective statements and loop statements were executed, another two states (true or false) are also specified. Control dependence is defined in [32], and the algorithm to compute the control dependence in [32] is also used in this paper. State dependency is proposed to describe the control flow dependence between statements with particular states.

**Definition 2 (State Dependency, SD):** For a two-tuple (nd,St), where nd is a node of a CFG, St is a set of states for the node nd. The i-th parent node of nd is denoted as paraᵢ(nd), and a set of states of paraᵢ(nd) is denoted as Stᵢ. The state dependency, which is denoted as SD(nd(st),paraᵢ(nd)(st’)), represents the dependency between the nd and paraᵢ(nd), where the execution state of nd is st, and the execution state of paraᵢ(nd) is st’.
st is one of the states in St, and st’ is one of the states in St’.

The control dependence path depicts the executed node with particular state when executing the test cases. As shown in Fig. 2, to the right of the program is the control dependence path with respect to each test case. Each column indicates a test case and its corresponding execution control dependence path, and the passed/failed result of each test case is shown at the bottom of each column. The state dependency probability of each statement is calculated according to CFG and control dependence paths.

1. For each statement, if its parent node is executed, the node has two states, which represent whether the statement is executed or not. The probability of the statement to be executed is calculated, that is P(node). Our technique assumes that the probability of program entry point to be executed is 1, and then the probability of the other statements to be executed should be calculated. The formula (1) means that P(node) is depending on its control dependent parent node P(para(node)) (m is the number of its parent nodes).

\[
P(node) = \frac{n(node)}{\sum_{i=1}^{m} n(para_i(node))} \times \sum_{i=1}^{m} P(para_i(node))
\]

2. When the selective statements and loop statements are executed, they have other two states (true or false), which determine the control flow path. For example, the “true” state for an if statement means that the program will execute the true-branch. The probabilities of both true and false state, which are denoted as P(node(true)) and P(node(false)) respectively, should be calculated.

\[
P(node(true)) = \frac{n(node(true))}{n(node)} \times P(node) = \frac{n(node(true))}{n(node(true)) + n(node(false))} \times P(node)
\]

\[
P(node(false)) = \frac{n(node(false))}{n(node)} \times P(node) = \frac{n(node(false))}{n(node(true)) + n(node(false))} \times P(node)
\]

P(para(node)) is the state dependency probability of its control dependent parent node with particular states, n(node) is the number of times the execution node is observed in control dependence paths, and n(para(node)) is the number of times its control dependent parent nodes are observed in control dependence paths. It is
worthwhile to note that if the node has several control dependent parent nodes, \( n(\text{para}(\text{node})) \) is the sum of all the control dependent parent nodes, \( n(\text{node}(true)) \) and \( n(\text{node}(false)) \) are the number of times the execution node is observed to be \( true \) and \( false \) state in control dependence paths, respectively.

The selective statements and loop statements have only two states (\( true \) or \( false \)) when they are executed, so

\[
P(\text{node}(true)) + P(\text{node}(false)) = P(\text{node})
\]

It is worthwhile to note that the back edge for loop structure is not considered in our approach. Moreover, we believe that the exit node of loop structure must be executed if this loop structure is executed, so the state dependency probability of exit node is equal to that of entry node.

We define State Dependency Probabilistic Model based on CFG and the states of each program element.

**Definition 3 (State Dependency Probabilistic Model, SDPM):** The state dependency probabilistic model is a triple \((C,S,R)\), where \( C \) is CFG of the program, \( S \) is the mapping from nodes of CFG to states, and \( R \) represents the probabilities between statements with particular states.

The process for establishing the state dependency probabilistic model is shown in Fig. 3. The algorithm is as follows:

1. The CFG of the program is constructed;
2. The test cases are executed after program instrumentation, and control dependence paths are obtained;
3. The state dependency probability of each node is calculated according to formula (1) to (4) based on the CFG and control dependence path.

To illustrate how to establish the state dependency probabilistic model, let us consider the program mid() and test cases shown in Fig. 2. For example, we choose three test cases \( \{t_1, t_2, t_3\} \) and calculate the state dependency probability of each statement. In Fig. 1, the probability of program entry point “int mid (int x, int y, int z)” is 1. According to the CFG of the program, statement 1 has only 1 parent node (node 0).
The number of times the node 1 has been executed is equal to that of its control dependent parent node 0. So \( P(1) = 1 \). In Fig. 2, node 2 is observed 3 times in control dependence paths of test cases \( \{ t_1, t_2, t_3 \} \), so \( n(2) = 3 \). The node 2 has only 1 parent node (node 1), and \( n(1) = 3 \). According to formula (1),

\[
P(2) = \frac{n(2)}{n(1)} \times P(1) = \frac{1}{1} \times 1 = 1
\]

Node 2 is a selective node, so

\[
P(2(true)) = \frac{n(2(true))}{n(2(true)) + n(2(false))} \times P(2) = \frac{1}{1+2} \times 1 = \frac{1}{3}
\]

(5)

According to formula (4),

\[
P(2(false)) = P(2) - P(2(true)) = 1 - \frac{1}{3} = \frac{2}{3}
\]

(6)

Node 3 can be executed only if node 2 gets a true state, so

\[
P(3) = P(2(true)) = \frac{1}{3}
\]

(7)

\[
P(3(false)) = \frac{n(3(false))}{n(3(true)) + n(3(false))} \times P(3) = \frac{1}{1+0} \times 1 = \frac{1}{3}
\]

(8)

\[
P(3(true)) = P(3) - P(3(false)) = \frac{1}{3} - \frac{1}{3} = 0
\]

(9)

The node 12 has multiple parent nodes (node 4, 6, 9, 11), so

\[
P(12) = \frac{n(12)}{\sum_{i=1}^{4} n(\text{para}_i(12))} \times \sum_{i=1}^{4} P(\text{para}_i(12))
\]

\[
= \frac{n(12)}{n(4) + n(6) + n(9) + n(11)} \times (P(4) + P(6) + P(9) + P(11))
\]

(10)

In Fig. 2, \( n(12) = 3 \), \( n(4) = 0 \), \( n(6) = 1 \), \( n(9) = 1 \), and \( n(11) = 1 \). According to Fig. 1, \( P(4) = P(3(true)) \), \( P(6) = P(5) = P(3(false)) \), \( P(9) = P(8(true)) \), \( P(11) = P(10) = P(8(false)) \), so \( P(4) + P(6) + P(9) + P(11) = P(3(true)) + P(3(false)) + P(8(true)) + P(8(false)) = P(3) + P(8) \);

\( P(3) = P(2(true)) \), \( P(8) = P(2(false)) \), \( P(4) + P(6) + P(9) + P(11) = P(2(true)) + P(2(false)) = P(2) \).

\[
P(12) = \frac{3}{0 + 1 + 1 + 1} \times P(2) = \frac{3}{3} \times 1 = 1
\]

(11)

According to the above method, we can obtain the state dependency probability table (SPT). In order to demonstrate the fault localization approach in section 3, the test
cases are divided into two groups, passed test cases and failed test cases. Table 1 and 2 are the SPT corresponding to passed test cases \{t_1, t_2, t_3\} and failed test cases \{t_4, t_5\}, respectively.

It is worthwhile to note that, the calculation method of state dependency probability for loop statements is the same for selective statements. For example, Fig. 4 shows the code segment and its corresponding CFG. The loop node 5 has two parent nodes (node 4 and 8). The edge 8->5 is the back edge for loop structure, so this edge is not considered. Here we suppose that the state dependency probability of node 4 is calculated by formula (1), and the value is 1, that is \(P(4)=1\). In Fig. 4, node 5 must be executed when its parent node 4 is executed, so \(P(5)=P(4)=1\). The node 5 has another two states (true or false). The “true” state means that the program will execute node 6, and the “false” state means that the program will execute node 9. Here we suppose that \{4,5(true),6(true),7,6(false),8,5(true),6(false),8,5(false),9,10\} is one of the control dependence paths. \(P(5(true))\) and \(P(5(false))\) can be calculated according to formula (2) and (3).

\[
P(5(true)) = \frac{n(5(true))}{n(5(true)) + n(5(false))} \times P(5) = \frac{2}{2+1} \times 1 = \frac{2}{3} \quad (12)
\]

\[
P(5(false)) = \frac{n(5(false))}{n(5(true)) + n(5(false))} \times P(5) = \frac{1}{2+1} \times 1 = \frac{1}{3} \quad (13)
\]

We consider that entry node and exit node for loop structure have the same state dependency probability, so \(P(9)=P(5)=1\). The statement 6 is in the true-branch of loop structure, so \(P(6)=P(5(true))=2/3\).

The node 6 is a loop statement, so \(P(6(true))\) and \(P(6(false))\) should be calculated.

\[
P(6(true)) = \frac{n(6(true))}{n(6(true)) + n(6(false))} \times P(6) = \frac{1}{1+2} \times \frac{2}{3} = \frac{2}{9} \quad (14)
\]

\[
P(6(false)) = \frac{n(6(false))}{n(6(true)) + n(6(false))} \times P(6) = \frac{2}{1+2} \times \frac{2}{3} = \frac{4}{9} \quad (15)
\]

The statement 7 is in the true-branch of loop structure, so \(P(7)=P(6(true))=2/9\).

\[
P(10) = \frac{n(10)}{n(9)} \times P(9) = \frac{1}{1} \times 1 = 1 \quad (16)
\]
3. Fault Localization based on SDPM

As mentioned above, if the state dependency is rarely executed in passed test cases, but frequently executed in failed test cases, this state dependency is more likely to be faulty. From this point of view, we proposed an automatic fault-localization approach based on SDPM to differentiate the state dependencies between passed and failed test cases. The steps of fault localization are shown in Fig. 5. The state dependency probabilistic models for passed and failed test cases are denoted as SDPM(true) and SDPM(false), respectively.

1. Execute passed and failed test cases so as to establish SDPM(true) and SDPM(false), respectively;

2. Calculate a suspicious score for each state based on SDPM(true) and SDPM(false). The concrete algorithm is as follows:

   (1) For selective node and loop node, the suspicious scores are calculated for both true state and false state, which are denoted as SUS_T and SUS_F, respectively.

   \[
   \text{SUS}_T = \frac{P_{\text{failed}}(\text{node}(true))}{P_{\text{passed}}(\text{node}(true))} \tag{17}
   \]

   \[
   \text{SUS}_F = \frac{P_{\text{failed}}(\text{node}(false))}{P_{\text{passed}}(\text{node}(false))} \tag{18}
   \]

   \[
   P_{\text{failed}}(\text{node}(true)) \text{ is the state dependency probability of the node with a true state when executing failed test cases. Similarly, } P_{\text{passed}}(\text{node}(true)) \text{ is the state dependency probability of the node with a true state when executing passed test cases.}
   \]

   (2) For node containing single state, the suspicious score is calculated according to formula (19):

   \[
   \text{SUS} = \frac{P_{\text{failed}}(\text{node})}{P_{\text{passed}}(\text{node})} \tag{19}
   \]

   \[
   P_{\text{failed}}(\text{node}) \text{ is the state dependency probability of the node when executing failed test cases.}
   \]

   (3) In formula (17), (18), and (19), if the state dependency probability corresponding to failed test cases is zero, the corresponding node will not be ranked, because that means the state dependency is not executed in failed test cases, so this state
dependency is less likely to be faulty.

(4) In formula (17), (18), and (19), the suspicious score is infinite if denominator is zero. It means that the state dependency is not executed in passed test cases and it is only executed in failed test cases, so this state dependency is more likely to be faulty. It is noteworthy that if the denominators of two nodes are both zero, we compare the value of numerator. The larger numerator means the state dependency is more frequently executed in failed test cases in runtime, so the corresponding statement is more likely to be a faulty statement.

3. Rank the statements according to their suspicious score. It is worthwhile to note that our approach rank selective statements and loop statements according to SUS$_T$ and SUS$_F$. In this study, we rank the selective statements and loop statements with higher suspicious score. For example, suppose node 5 is a loop statement, and SUS$_T$ = 0.5, SUS$_F$ = 0.3. The final suspicious score of this node is 0.5.

To illustrate how to locate the faults based on SDPM, let us consider the example program mid() (Fig. 2), we can calculate the suspicious score of bug node 3 according to Table 1 and 2. The node 3 is a selective statement, so we calculate SUS$_T$ and SUS$_F$, respectively.

\[
\text{SUS}_T = \frac{P_{\text{failed}}(3(\text{true}))}{P_{\text{passed}}(3(\text{true}))} = \frac{1/2}{0} = \infty
\] (20)

\[
\text{SUS}_F = \frac{P_{\text{failed}}(3(\text{false}))}{P_{\text{passed}}(3(\text{false}))} = \frac{1/2}{1/3} = 1.5
\] (21)

In formula (20), the denominator $P_{\text{passed}}(3(\text{true}))$ is 0. This means that the state dependency only exists in failed test cases, so node 3 with a true state is very suspicious.

4. Experiments and Analysis

4.1 Experimental Setup

In our experiment, we use Siemens programs [39] and four UNIX programs as the subject programs to evaluate the effectiveness of our approach, because these programs have been widely used in fault-localization studies [10,13,19,25,33,38]. The programs and test cases used in this study are downloaded from
The Siemens programs are written in C, and they are a suite of seven small programs, including print_tokens, print_tokens2, replace, schedule, schedule2, tcas and tot-info. Each program has more than 1,000 test cases. The UNIX programs are real-life, including flex, grep, gzip and sed. These UNIX programs have more lines of code and less test cases than Siemens programs. The subject UNIX programs and the results of CBI, SOBER, Jaccard, SBI, Tarantula, and CP on UNIX programs are provided by Zhang [30]. Table 3 shows the characteristics of Siemens programs and Table 4 shows the characteristics of UNIX programs. In order to compare with previous methods [30], some faulty versions are excluded.

1. The versions which have no failed test cases.
2. The versions that fails for more than 20% of the test cases.
3. The versions which have segmentation fault.
4. Zhang et al. [30] excluded some faulty versions that are not supported by their experimental environment, and we also excluded these versions to compare our experimental results with CP.

For Siemens programs, version 9 of schedule 2, version 27 of replace, versions 5, 6 and 9 of schedule are removed. After removing these versions, 126 versions of Siemens programs and 110 versions of UNIX programs are left to evaluate the effectiveness of our approach. Each faulty version contains exactly one fault, although the faults may span multiple statements or even functions. In our experiment, a program instrumentor is implemented via Yacc. It can construct CFG for program and record control dependence paths for every test case. Before applying our strategy, we have done some checked tests for all test cases, whose results are passed or failed. The number of these passed or failed tests can be arbitrary in practice. We apply the passed and failed test suite as inputs to individual subject programs. And then our approach is used to locate faults. In order to compare our approach with previous researches, we also use the metric score [10,19,32] to calculate the percentage of the program that need to be examined to find a faulty statement.

4.2 Experimental results and analysis

SBI, Tarantula, Jaccard, CP, and our approach provide a ranked list of all
statements. In order to compare with previous researches [30], we also use the metric score to calculate the percentage of the program that need to be examined to find a faulty statement. CBI and SOBER only locate the faults relevant to predicates, so T-score was used to evaluate the fault-localization effectiveness. The T-score estimates the percentage of code that has been examined in order to locate the fault.

\[
T = \left( \frac{|N_{\text{examined}}|}{|N|} \right) \times 100\%
\]  

\( |N_{\text{examined}}| \) is the number of nodes examined to find faulty node and \( |N| \) is the size of the program dependence graph. It has been proven that the “top-5 T-score” strategy gives the highest performance for CBI and SOBER, so the top-5 T-score results of CBI and SOBER were reported by Zhang et al. [30]. The results are also used in this paper.

Fig. 6 shows the experimental results on Siemens programs. We obtained the fault-localization results for SBI, SOBER, and Ochiai from published papers [20, 25, 26]. It shows the percentage of faults that can be located when a certain percentage of code is examined. The x-axis means the percentage of code that needs to be examined in order to locate the fault. The y-axis means the percentage of fault located, and it is calculated according to the percent of total for each program. The lower the percentage of code examined, the higher the effectiveness of the fault-localization technique is. We find that our approach can locate more faults than the other methods in every range. Tarantula and SOBER perform better than CP by checking less than 40% code, and Ochiai performs better than CP by checking less than 35% code. However, CP catches up with Tarantula gradually. CP performs worst by checking less than 5% code.

Fig. 7 shows the results of each fault-localization technique on each subject Siemens program. We observe that the effectiveness of our approach, Tarantula and CP are very similar for program tcas and tot-info (Fig. 7(d), Fig. 7(g)). For program replace (Fig. 7(a)), our approach can locate more faults than CP, and it has similar fault-localization effectiveness with Tarantula. For the other programs, our method obviously outperforms CP and Tarantula.

Fig. 8 shows the experimental results on UNIX programs. We observe from the figure that on the whole, CP and our approach have the best effectiveness on fault
localization, Tarantula, SBI, and Jaccard take second place, CBI and SOBER are the worst. The curves of Tarantula, SBI, and Jaccard almost completely overlap, that means the effectiveness of these methods are very close. CP and our approach can locate more faults than CBI and SOBER. The overall efficiency of our approach in the range of 10%-30% of analyzed source code is higher than the other methods.

Fig. 9 shows the results on the first 20% code examination range for each of the subject UNIX program because the overall effectiveness can not meaningfully conclude the results from outliner segments. We find that our approach can locate more faults than the other methods by checking less than 1% code. When a programmer examine up to 1% of the code, our approach can discover 28.18% of all the faults; Tarantula, SBI, and Jaccard can locate 26.36% of all the faults; CP and CBI can only do so for about 18.18%, 5.45%, respectively; while SOBER can not locate any fault. CP can locate more faults than our approach in the range from 2% to 10% of the code affordable to be examined. However, our method catches up and exceeds CP by checking more than 10% code (from 10% to 20%).

Fig. 10 shows the results of each fault-localization technique on each of the subject UNIX program. For subject program flex (Fig. 10(a)), the results of our method consistently outperform those of other methods. For program grep (Fig. 10(b)), CP seems to have better results than our approach in the entire range of the first quarter (from 0% to 25%) of the code examination effort, but our approach catches up by checking more than 25% code. For program gzip (Fig. 10(c)), the situation is reverse. Thus, it is difficult to tell which one is better. For program sed (Fig. 10(d)), CP seems to have better results than our approach, but the advantage is not obvious.

We use statistics metrics to compare different techniques. Table 5 lists out the minimum (min), maximum (max), medium, mean, and standard derivation (stdev) of the effectiveness of these techniques. The smaller the magnitude, the better is the effectiveness. The results show that our approach gives the smallest value among seven techniques, so our approach is more effective on locating faults.

Table 6 shows the difference in effectiveness between our approach and each peer technique. For example, the cell in column “our approach-CP” and row “<-1%” means
that for 44 (40.00%) of the 110 faulty versions, the code examination effort of using our approach to locate a fault is less than that of CP by more than 1%. Similarly, for the row “>1%”, only 33 (30.00%) of the faulty versions, the code examination effort of our approach is greater than that of CP by more than 1%. For 33 (30.00%) of the faulty versions, the effectiveness between our approach and CP cannot be distinguished at the 1% level. Therefore, we deem that at the 1% level, the probability of our approach performing better than CP on these subject programs is higher than that of CP performing better than our approach. We further vary the level from 1% to 5% and 10% to produce the complete table. The experimental results show that the probability of our approach performing better than its peer technique is consistently higher than that for the other way round.

The American Psychological Association strongly encourages reporting effect sizes in original units: “Effect sizes may be reported in original units…and are most easily understood in original units. [53]”. Cohen’s $d$ is a standardized effect size measure [54], which is used in this paper to judgment the practically significant of our approach. For Cohen’s $d$, we can obtain $d$ from formula (23) and (24):

$$d = \frac{\bar{X}_1 - \bar{X}_2}{S_p} \quad \text{(23)}$$

$$S_p = \sqrt{\frac{(n_1-1)S_1^2 + (n_2-1)S_2^2}{n_1 + n_2 - 2}} \quad \text{(24)}$$

$\bar{X}_1$ is the sample mean of group 1, $\bar{X}_2$ is the sample mean of group 2, $S_p$ is the pooled standard deviation, $n_1$ is the sample size for group 1, $n_2$ is the sample size for group 2, $S_1^2$ is the standard derivation of group 1, $S_2^2$ is the standard derivation of group 2. Normally, the effect size is small if $0 < d < 0.2$, the effect size is medium if $0.2 < d < 0.8$, and the effect size is large if $d > 0.8$. Cohen points out that a large effect will not necessarily be clinically important, and sometimes small effects are clinically important in some fields.

The effect sizes are reported in table 6. The effect size of our approach-CP is relatively small, our approach-Tarantula and our approach-SBI have similar effect size,
and the effect size of our approach-SOBER is relatively large. It is worthwhile to note that it is difficult to rise up the rank of faulty statement in the field of fault localization, and we believe that the effects are still clinically important, even though the effect size is relatively small.

In order to further investigate the effect of our approach compared with the previous method, a metric \( \text{EffectivenessChange} \) is defined to evaluate the fault-localization effectiveness.

\[
\text{EffectivenessChange} = \frac{\text{PreRank} - \text{OurRank}}{\text{number of executable statements}} \times 100\% 
\]

\( \text{PreRank} \) is the rank of a faulty statement in previous method (such as Tarantula, SBI, SOBER). \( \text{OurRank} \) is the rank of a faulty statement in our method. Clearly, upper score indicates our method has a better effect on the fault localization. Positive \( \text{EffectivenessChange} \) indicates our method is more effective than previous method, while negative \( \text{EffectivenessChange} \) indicates that our method is less effective than previous method. Boxplot\(^\dagger\) is used to depict the distribution of \( \text{EffectivenessChange} \).

For example, Fig. 11(a) shows that most of boxplots are above X axis for program flex and gzip, that indicates the fault-localization effectiveness are generally improved compared with CP. The boxplots of program sed are very narrow, which indicates that the \( \text{EffectivenessChange} \) in total faulty versions is similar. Most of boxplots are below X axis for program grep, that indicates our approach reduces the fault-localization effectiveness compared with CP. None of the methods can locate all types of faults, and each method has its own advantage and proper scope. Thus, not all the boxplots are above X axis.

In practice, it is difficult to obtain multiple passed and failed test cases. The goal of our first study is to determine the effectiveness of our method using only several passed and failed test cases for a single-fault program. Fig. 12 shows the cumulative percentage of all ranked sets of statements in each score range computed by our method and Tarantula using different numbers (1, 3, 5) of failed test cases on Siemens programs.

\( ^\dagger \) A boxplot is a standard statistical device for representing data sets. It consists of five important sample percentiles: the sample minimum, the lower quartile, the median, the upper quartile and the sample maximum. The box’s height spans the central 50% of the data and its upper and lower ends mark the upper and lower quartiles. The middle of the three horizontal lines within the box represents the median.
The number of passed test cases used in this study is five. The results show that our approach performs better than Tarantula when only a few passed test cases are available, and both techniques achieve better results as more failed test cases are available.

We also examined the effectiveness of our method using only one failed test case at a time and analyzed it with the help of multiple (1, 3, 5) passed test cases on Siemens programs. Fig. 13 shows that our approach and Tarantula can locate more faults as more passed test cases are available, and our method performs better than Tarantula when only a few passed test cases are available.

In the experiment, we also measure the ability of our approach when dealing with the programs containing multiple faults on Siemens programs. We only combined two single-fault faulty versions if their version numbers are consecutive. It is worthwhile to note that our approach rank the statements by using the faulty statement with higher rank. Fig. 14 shows the effectiveness of our method and Tarantula using five passed test cases and different numbers (1, 3, 5, 10) of failed test cases when dealing with multiple faults. The results show that our approach performs better than Tarantula when dealing with multiple faults. The experimental results show that the fault-localization effectiveness of our method can not be improved when using 3 failed tests compared with using 1 failed test case. The fault-localization effectiveness is slightly improved by using 5 failed test cases, and it is improved drastically when using 10 failed test cases. Tarantula achieves worse results first and then become better and better as the number of analyzed failed test cases increase.

Table 7 and 8 summarize the mean time of our approach on Siemens programs and UNIX programs. The experiment environment is Intel(R) Core(TM) i3-2350M CPU @2.30GHz, Memory 4.00GB. All the timings are in seconds. The columns show the programs, the average time taken to program instrumentation and control dependence path, the average time taken to build the SDPM, and the average time to locate faults. The results show that the construction of SDPM spends most of time in our approach. Our approach analyzes the behavior state of each statement and the state dependence information between program elements, which are not considered in statement-level technique, so a typical time needed for statement-level technique to profile data and
rank statements seems to be one to two orders of magnitude lower than our approach.

4.3 Discussion

CBI and SOBER are predicate-level techniques. CBI only captures predicate coverage information, and it only analyzes whether a predicate has been executed. SOBER can distinguish the number of times the predicate is evaluated to be true or false, respectively. Tarantula, SBI, and Jaccard are statement-level techniques. They can rank all executable statements, but they only analyze the coverage information rather than execution path, which is not enough to analyze program execution behavior. Our experimental results show that CP and our approach can locate more faults than CBI and SOBER, that means control dependence-level techniques are more effective than basic predicate-level techniques. The reason should be that control dependence-level techniques can capture richer information about program executions than basic predicate-level techniques. CP and our approach analyze the control dependence between program elements. We call these control dependence-level techniques. It is worthwhile to note that none of the methods can locate all types of faults, and each method has its own advantage and proper scope. CP investigates how each program entity contributes to failures by abstractly propagating infected program states to its adjacent basic blocks through control flow edges. The aim is to investigate the propagation of infected program states among program entities, and the advantage is not obvious when locating the faults irrelevant to fault propagation. Moreover, the execution state is an important factor in the program analysis, which determines the concrete execution path through a program. However, the execution state is not analyzed by CP, that would reduce the accuracy of fault localization. Our approach uses path profiles to capture the behavior state information of each program element based on the analysis of control flow dependence, and then locate the faults. The aim is to investigate the differences of the state dependency in passed and failed test cases. Our approach can perform better than CP when the faults are irrelevant to the propagation of errors. For example, the fault in Fig. 15 is irrelevant to failure propagation. CP first ranks statements 7 and 8 as the most suspicious statements, but these statements are not real faults. In our approach, statements 3 and 4 are identified as the most suspicious
statements, and the faulty state is also provided. From Fig. 15 we can see that the suspicious score of statement 3 is infinite, and the faulty state is “true”. Fig. 15 shows Tarantula seems also effective enough to locate the bug. It is worthwhile to note that Tarantula is a statement-level technique. The concrete execution path information can not be analyzed in Tarantula, which affects the effectiveness of fault localization. For example, t₁ and t₂ are test cases of the program in Fig. 16, and the corresponding control dependence path of t₁ and t₂ are also provided. Execution traces are represented by ●. Fig. 16 shows that our approach obviously outperforms Tarantula. This is because the execution path can capture more program execution behavior for fault localization than basic coverage information.

The experimental results show that our approach achieves better results as more failed test cases are available. We observe that in the experiment, the suspicious score of faulty statement increases with the number of failed test cases. Our approach can locate more faults as more passed test cases are available, it is because that the suspicious score of some correct statements decreases with the number of passed test cases, and the suspicious score of faulty statement increases with the number of passed test cases at the same time.

4.4 Threats to Validity

In this section, we discuss the threats to internal, external, and construct validity of our experiment.

Threats to internal validity mainly come from the incorrect program implementation. To overcome these threats, we compared manually generated control dependence paths of smaller subjects to their control dependence paths generated automatically by our instrumentor to ensure that the control dependence paths match (which they did).

Threats to external validity concern the adequacy of the data set. In this experiment, we evaluated the effectiveness of our fault-localization approach using Siemens programs and several UNIX programs, and the Siemens programs are too small in terms of program sizes and all the faults are artificially seeded by researchers. Thus, we are unable to definitively state that our findings will hold for programs in general. To
address some of these uncertainties, we performed evaluation on the programs containing multiple faults to demonstrate how this factor affected the results. We also performed evaluation on a varying number of passed and failed test cases. Further applications of our approach to medium-to-large-sized real-life programs would strengthen the external validity of our work. Moreover, several passed and failed test cases are chosen to investigate the effect of varying number of test cases on fault-localization effectiveness. In previous research work, some researchers [40-51] focused on investigating how the test cases affect the fault-localization effectiveness. We also proposed a test cases reduction approach in our previous work to provide suitable test cases input for fault localization [52]. In this paper, the test cases are randomly chosen, and we did not measure the fault-localization effectiveness of different test cases selection strategies. We will address this threat in future work.

Another threat to external validity arises when comparing our results on a par with previous works. Our method and previous approaches use different platforms, which may generate different results. In this connection, the comparison should be interpreted carefully.

Threats to construct validity concern the appropriateness of the metrics used in our evaluation. The metric score is used to determine the fault localization effectiveness of our approach. However, it is difficult to determine whether it conforms to the way in which programmers perform fault localization. Therefore, more studies are required to determine the appropriateness of the metric for evaluating fault-localization techniques.

5. Related work
5.1 Fault localization

To date, there are four main approaches to fault localization. The first approach is program slicing, including static slicing [2], dynamic slicing [3-8] and execution slicing [9]. The static slicing of an incorrect variable at an execution point includes all those program statements which possibly could influence the value of the variable at that point. It does not make any use of the input values that reveal the fault. By contrast, the dynamic slicing of an incorrect variable at a program point is the set of executed
statements which actually affect the value of the variable at the given program point under some execution. The execution slicing is the set of code executed by a given test cases, and the construction of execution slicing is easier than that of static slicing and dynamic slicing. By studying the program slicing of the incorrect value, a programmer can eliminate the irrelevant value and narrowing search area to detect the faulty statements. However, there may still be too much code that needs to be examined. Previous research [13] shows that identifying the faulty code from the set of statements in the slice still requires nontrivial human effort.

The second approach to the fault localization is using state altering [10-13]. This method finds a predicate that causes the program to produce incorrect results. By modifying the state of the predicate and re-executing the program, the predicate’s outcome is switched to produce the desired change in control flow, and the cause of the bug can be identified. The major problem of this approach is that the search space of potential state changes is extremely large. Another problem is that not all of the faulty predicates can be identified, because following a predicate switch, some predicates’ outcome can not be analyzed.

The third approach is using model-based [14-18] methods to identify incompatibilities, unexpected interactions, undesirable behaviors and so on. It infers object behavioral models from execution traces, and detects behavior incompatibilities by contrasting this model with the behavior the components display when reused in new contexts. However, such an approach is computationally much more demanding and may still produce a large output that lacks ranking information. As a result of its high complexity and huge overhead of inferring behavior models, the application of this approach is limited.

The fourth approach is statistical analysis [19-26]. This approach locates fault-relevant statements by comparing the statistical differences of program elements in passed and failed test cases. Typical examples of such techniques are CBI [20], SOBER [24,25] and Tarantula [19], which are relevant to our approach. These techniques have been explained in and compared with our approach.

5.2 Test cases selection
In order to improve the effectiveness of fault localization, some test cases are selected to provide suitable test cases inputs. Thus, the effect of test-suite selection on fault-localization effectiveness has been widely studied. In previous research work, some researchers focused on proposing different approaches to select test cases, and then investigated how the selective test-suite affect the fault-localization effectiveness according to their own experimental artifacts. Offutt et al. [40] applied heuristics to find the minimal test cases. Orso et al. [41] presented MINTS framework to find an optimal solution for different test cases minimization problems, and they pointed out that their approach was as efficient as heuristic approaches. However, the above methods can not analyze the impact of test-suite minimization on the fault-localization effectiveness. Wong et al. [42] investigated the impact of test cases size minimization on fault localization. They assumed that block minimized test cases had a size advantage with almost the same fault-localization effectiveness. Rothermel et al. [43] also studied the impact of test-suite minimization on the fault localization, and they pointed out that the fault localization capabilities of test suite were severely compromised by minimization. Baudry et al. [44] defined a dynamic basic block as a set of statements that was covered by the same test cases, and then investigated the relationship between the number of the dynamic basic block and fault-localization effectiveness. Zhang et al. [45] proposed the concept of relative redundancy for test-suite reduction for the first time, and then proposed a new evaluation technique to balance the uneven distribution in the reduced test-suite. Chen et al. [46] assumed that some faults had relationship with interaction of requirements and these interactions should be considered in the test-suite reduction. Hao et al. [47-48] proposed several statement-based reduction strategies to acquire high-statement-coverage test-suite based on the execution traces of test runs using the test inputs, and they assumed that test cases redundancy or similarity decreased the effectiveness of fault localization. However, the experimental results of Yu et al. [49] showed that additional redundancy did not reduce the fault-localization effectiveness. They proposed vector-based reduction techniques and found that statement-based reduction strategy provided much greater reduction of the test-suite than vector-based reduction, but vector-based reduction was more effective on fault localization.
The execution path can capture more information about program execution behavior than statement coverage. Previous research indicated that path-based fault localization technique was more effective than coverage-based fault localization because the semantic of a program could be analyzed [21]. In all above test cases selection approaches, researchers only analyzed coverage information of test cases. Unfortunately, there was not any study that investigates the test-suite reduction based on the execution path. Researchers used their own experimental artifacts to verify the effect of test-suite reduction approach on fault-localization effectiveness. Our previous test cases selection approach [52] was path-based, and it analyzed the coverage information as well as the concrete execution path of test cases. We also proposed loop standardization based on the execution path information to improve the distribution evenness of execution paths. The experimental results show that our selective test cases can improve fault-localization effectiveness.

6. Conclusion

In this paper, we proposed a state dependency probabilistic model for fault localization. Compared with previous studies, our approach not only investigates the impact of execution control flow in runtime, but also analyzes the state dependencies of program elements. The proposed model can capture the behavior state information during program execution, and the fault-localization approach differentiates the state dependencies in passed and failed test cases. Experimental results show that our approach consistently outperforms the other evaluated techniques in terms of effectiveness in fault localization on Siemens programs. The results also show that the SDPM can be an effective approximate model for representing behaviors of a program for fault localization. Moreover, our approach is highly effective in fault localization even when very few test cases are available. It also performs well in the presence of multiple faults.

Acknowledgements

This research is supported by the National Natural Science Foundation of China (Grant No. 61173021 and 61202092) and the Research Fund for the Doctoral Program of Higher Education of China (Grant No. 20112302120052 and 20092302110040). We
are extremely grateful to Zhang Zhenyu for the subject programs and their experimental results, and to anonymous reviewers for their suggestions for improvement.
References


International Conference on Software Maintenance (ICSM'98), 1998.


Gong Dandan, born in 1982. PhD candidate in the Computer Science Department of Harbin Institute of Technology. Her current research interests include software bug detection, program analysis and software engineering.

Wang Tiantian, born in 1980. Lecturer in the Computer Science Department of Harbin Institute of Technology. Her current research interests include software engineering, program analysis and computer aided education.

Su Xiaohong, born in 1966. She has been professor of Harbin Institute of Technology since 2004. She is a senior member of China Computer Federation. Her main research interests include software bug detection, graphics and image processing, information fusion and intelligent computation.

Ma Peijun, born in 1963. Professor and PhD supervisor. His main research interests include space computation, information fusion, color matching, image processing and intelligent control.

Wang Yu, born in 1989. Master candidate in the Computer Science Department of Harbin Institute of Technology. His current major research direction is software bug locating.
Captions:

Fig. 1 The CFG of example program mid().
Fig. 2 Program mid() and its execution control dependence paths.
Fig. 3 Steps in the construction of state dependency probabilistic model.
Fig. 4 The CFG for loop structure.
Fig. 5 The steps of fault localization based on SDPM.
Fig. 6 Overall effectiveness comparison on Siemens programs.
Fig. 7 Effectiveness on individual programs on Siemens programs.
Fig. 8 Overall effectiveness comparison on UNIX programs.
Fig. 9 Overall results in zoom-in range of [0%, 20%] on UNIX programs.
Fig. 10 Effectiveness on individual programs on UNIX programs.
Fig. 11 Comparison of the distribution of EffectivenessChange between our approach and previous method.
Fig. 12 Effectiveness of fault localization using different numbers of failed test cases on Siemens programs.
Fig. 13 Effectiveness of fault localization using different numbers of passed test cases on Siemens programs.
Fig. 14 Effectiveness of fault localization on Siemens programs in the presence of multiple faults.
Fig. 15 Fault localization comparison between CP, Tarantula, and our approach.
Fig. 16 Fault localization comparison between Tarantula and our approach.

Table 1. The SPT of the passed test cases \{t_1, t_2, t_3\}.
Table 2. The SPT of the failed test cases \{t_4, t_5\}.
Table 3. The Siemens programs.
Table 4. The UNIX programs.
Table 5. Statistics of effectiveness.
Table 7. Efficiency of our approach in seconds on Siemens programs.
Table 8. Efficiency of our approach in seconds on UNIX programs.
Fig. 1 The CFG of example program `mid()`. 

```c
0 int mid(int x, int y, int z) {
1    int m = z;
2    if (y < z);
3        if (x > y); // **bug**
4        m = y;
5    else if (x < z)
6        m = x;
7    else
8        if (x > y)
9            m = y;
10       else if (x < z)
11       m = x;
12    return m;
12 return m
```
Fig. 2 Program mid() and its execution control dependence paths.

```plaintext
int mid() {
    int x, y, z;

1: int m = z;
   t1  t2  t3  t4  t5
   2,3,1 3,2,1 1,1,2 1,2,3 3,1,2

2: if (y < z)
   2(false) 2(false) 2(true) 2(true) 2(true)

3: if (x > y)  //***bug***
   3(false) 3(false) 3(true)

4: m = y;
   4

5: else if (x < z)
   5(true) 5(true)

6: m = x;
   6  6

7: else

8: if (x > y)
   8(false) 8(true)

9: m = y;
   9

10: else if (x > z)
    10(true)

11: m = x;
    11

12: return m;
}
```

Passed / Failed Status

<table>
<thead>
<tr>
<th></th>
<th>t1</th>
<th>t2</th>
<th>t3</th>
<th>t4</th>
<th>t5</th>
</tr>
</thead>
<tbody>
<tr>
<td>P</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>P</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>P</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>F</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>F</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Fig. 3 Steps in the construction of state dependency probabilistic model.
Fig. 4 The CFG for loop structure.
Fig. 5 The steps of fault localization based on SDPM.
Fig. 6 Overall effectiveness comparison on Siemens programs.
Fig. 7 Effectiveness on individual programs on Siemens programs.
Fig. 8 Overall effectiveness comparison on UNIX programs.
Fig. 9 Overall results in zoom-in range of [0%, 20%] on UNIX pro.
Fig. 10 Effectiveness on individual programs on UNIX programs.
Fig. 11 Comparison of the distribution of EffectivenessChange between different approaches.
Fig. 12 Effectiveness of fault localization using different numbers of failed test cases.
Fig. 13 Effectiveness of fault localization using different numbers of passed test cases.
Fig. 14 Effectiveness of fault localization on Siemens programs

(a) 1 failed test case
(b) 3 failed test cases
(c) 5 failed test cases
(d) 10 failed test cases
<table>
<thead>
<tr>
<th>Statement</th>
<th>CP</th>
<th>Our approach</th>
<th>Tarantula</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>suspicious score</td>
<td>rank</td>
<td>suspicious score</td>
</tr>
<tr>
<td>1: int m = z;</td>
<td>-1.48</td>
<td>12</td>
<td>1</td>
</tr>
<tr>
<td>2: if ( y &lt; z )</td>
<td>-1.48</td>
<td>12</td>
<td>3(true)</td>
</tr>
<tr>
<td>3: if ( x &gt; y )</td>
<td>1.04</td>
<td>3</td>
<td>∞(true)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>∞</td>
</tr>
<tr>
<td>4: m = y;</td>
<td>1</td>
<td>4</td>
<td>1.5(true)</td>
</tr>
<tr>
<td>5: else if ( x &lt; z )</td>
<td>0.2</td>
<td>6</td>
<td>5</td>
</tr>
<tr>
<td>6: m = x;</td>
<td>0.2</td>
<td>6</td>
<td>5</td>
</tr>
<tr>
<td>7: else</td>
<td>2</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>8: if ( x &gt; y )</td>
<td>2</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>9: m = y;</td>
<td>-1</td>
<td>10</td>
<td>0</td>
</tr>
<tr>
<td>10: else if ( x &gt; z )</td>
<td>-1</td>
<td>10</td>
<td>0</td>
</tr>
<tr>
<td>11: m = x;</td>
<td>-1</td>
<td>10</td>
<td>0</td>
</tr>
<tr>
<td>12: return m; }</td>
<td>-0.5</td>
<td>7</td>
<td>1</td>
</tr>
</tbody>
</table>
Fig. 16 Fault localization comparison between Tarantula and our

<table>
<thead>
<tr>
<th>statement</th>
<th>test cases</th>
<th>Tarantula</th>
<th>our approach</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>t₁</td>
<td>t₂</td>
<td>suspicious score</td>
</tr>
<tr>
<td>1: ...</td>
<td>●</td>
<td>●</td>
<td>0.5</td>
</tr>
<tr>
<td>2: while ...</td>
<td>●</td>
<td>●</td>
<td>0.5</td>
</tr>
<tr>
<td>3: { if ...</td>
<td>●</td>
<td>●</td>
<td>0.5</td>
</tr>
<tr>
<td>4: ... <em><strong>bug</strong></em></td>
<td>●</td>
<td>●</td>
<td>0.5</td>
</tr>
<tr>
<td>5: else if ...</td>
<td>●</td>
<td>●</td>
<td>0.5</td>
</tr>
<tr>
<td>6: ... }</td>
<td>●</td>
<td>●</td>
<td>0.5</td>
</tr>
<tr>
<td>7: }</td>
<td>●</td>
<td>●</td>
<td>0.5</td>
</tr>
<tr>
<td>Pass / Fail Status</td>
<td>P</td>
<td>F</td>
<td></td>
</tr>
</tbody>
</table>

control dependence path of passed test case t₁:
{1,2(true),3(true),4,5(false),2(true),3(false),5(true),6,2(false),7}

control dependence path of failed test case t₂:
{1,2(true),3(true),4,5(false),2(true),3(true),4,5(false),2(true),3(false),5(true),6,2(false),7}
Table 1. The SPT of the passed test cases \{t_1, t_2, t_3\}.

<table>
<thead>
<tr>
<th>node</th>
<th>P(state)</th>
<th>true</th>
<th>false</th>
<th>node</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td></td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>1/3</td>
<td>2/3</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td>1/3</td>
<td>1/3</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>1/3</td>
<td>0</td>
<td>1/3</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
<td>1/3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>1/3</td>
<td>1/3</td>
<td>2/3</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td></td>
<td>1/3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>1/3</td>
<td>0</td>
<td>1/3</td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>1/3</td>
<td>1/3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>12</td>
<td></td>
<td>1</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

TABLE 1 The SPT of the passed test cases \{t_1, t_2, t_3\}.
Table 2. The SPT of the failed test cases \{t_4, t_5\}.

<table>
<thead>
<tr>
<th>P(state) node</th>
<th>true</th>
<th>false</th>
<th>node</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>1/2</td>
<td>1/2</td>
<td>1/2</td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td>1/2</td>
<td>1/2</td>
</tr>
<tr>
<td>4</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>5</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>6</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>8</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>9</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>11</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>12</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

TABLE 2 The SPT of the failed test cases \{t_4, t_5\}.
Table 3. The Siemens programs.

<table>
<thead>
<tr>
<th>Program</th>
<th>Description</th>
<th>Faulty versions</th>
<th>Lines of code</th>
<th>Test cases</th>
</tr>
</thead>
<tbody>
<tr>
<td>print_tokens</td>
<td>lexical analyzer</td>
<td>7</td>
<td>472</td>
<td>4130</td>
</tr>
<tr>
<td>print_tokens2</td>
<td>lexical analyzer</td>
<td>10</td>
<td>399</td>
<td>4115</td>
</tr>
<tr>
<td>replace</td>
<td>pattern replacement</td>
<td>31</td>
<td>512</td>
<td>5542</td>
</tr>
<tr>
<td>schedule</td>
<td>priority scheduler</td>
<td>5</td>
<td>292</td>
<td>2710</td>
</tr>
<tr>
<td>schedule2</td>
<td>priority scheduler</td>
<td>9</td>
<td>301</td>
<td>2650</td>
</tr>
<tr>
<td>tcas</td>
<td>altitude separation</td>
<td>41</td>
<td>141</td>
<td>1608</td>
</tr>
<tr>
<td>tot-info</td>
<td>information measure</td>
<td>23</td>
<td>440</td>
<td>1052</td>
</tr>
</tbody>
</table>
Table 4. The UNIX programs.

<table>
<thead>
<tr>
<th>Program</th>
<th>Real-life versions</th>
<th>Description</th>
<th>Faulty versions</th>
<th>Lines of code</th>
<th>Test cases</th>
</tr>
</thead>
<tbody>
<tr>
<td>flex</td>
<td>2.4.7-2.5.4</td>
<td>lexical parser</td>
<td>21</td>
<td>8571-10124</td>
<td>567</td>
</tr>
<tr>
<td>grep</td>
<td>2.2-2.4.2</td>
<td>text processor</td>
<td>17</td>
<td>8053-9089</td>
<td>809</td>
</tr>
<tr>
<td>gzip</td>
<td>1.1.2-1.3</td>
<td>compressor</td>
<td>55</td>
<td>4081-5159</td>
<td>217</td>
</tr>
<tr>
<td>sed</td>
<td>1.18-3.02</td>
<td>text processor</td>
<td>17</td>
<td>4756-9289</td>
<td>370</td>
</tr>
</tbody>
</table>

TABLE 4 The UNIX programs.
Table 5. Statistics of effectiveness.

<table>
<thead>
<tr>
<th></th>
<th>our approach</th>
<th>CP</th>
<th>Tarantula</th>
<th>SBI</th>
<th>Jaccard</th>
<th>CBI</th>
<th>SOBER</th>
</tr>
</thead>
<tbody>
<tr>
<td>min</td>
<td>0.01%</td>
<td>0.01%</td>
<td>0.01%</td>
<td>0.01%</td>
<td>0.01%</td>
<td>0.26%</td>
<td>2.97%</td>
</tr>
<tr>
<td>max</td>
<td>91.55%</td>
<td>93.55%</td>
<td>97.50%</td>
<td>97.50%</td>
<td>97.50%</td>
<td>100.00%</td>
<td>100.00%</td>
</tr>
<tr>
<td>medium</td>
<td>10.14%</td>
<td>11.67%</td>
<td>12.86%</td>
<td>12.48%</td>
<td>12.48%</td>
<td>40.74%</td>
<td>42.84%</td>
</tr>
<tr>
<td>mean</td>
<td>16.41%</td>
<td>17.98%</td>
<td>19.63%</td>
<td>19.74%</td>
<td>19.26%</td>
<td>40.55%</td>
<td>42.96%</td>
</tr>
<tr>
<td>stdev</td>
<td>20.01%</td>
<td>20.92%</td>
<td>22.47%</td>
<td>22.63%</td>
<td>22.39%</td>
<td>27.89%</td>
<td>23.98%</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Difference (percentage difference)</th>
<th>Our approach-CP</th>
<th>Our approach-Tarantula</th>
<th>Our approach-SBI</th>
<th>Our approach-Jaccard</th>
<th>Our approach-CBI</th>
<th>Our approach-SOBER</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt; -1%</td>
<td>44 (40.00%)</td>
<td>52 (47.27%)</td>
<td>53 (48.18%)</td>
<td>52 (47.27%)</td>
<td>82 (74.55%)</td>
<td>93 (84.55%)</td>
</tr>
<tr>
<td>-1% to 1%</td>
<td>33 (30.00%)</td>
<td>27 (24.55%)</td>
<td>28 (25.46%)</td>
<td>28 (25.46%)</td>
<td>5 (5.55%)</td>
<td>2 (1.82%)</td>
</tr>
<tr>
<td>&gt; 1%</td>
<td>33 (30.00%)</td>
<td>31 (28.18%)</td>
<td>29 (26.36%)</td>
<td>30 (27.27%)</td>
<td>23 (20.90%)</td>
<td>15 (13.63%)</td>
</tr>
<tr>
<td>&lt; -5%</td>
<td>26 (23.64%)</td>
<td>38 (34.55%)</td>
<td>38 (34.55%)</td>
<td>37 (33.64%)</td>
<td>76 (69.09%)</td>
<td>87 (79.09%)</td>
</tr>
<tr>
<td>-5% to 5%</td>
<td>65 (59.09%)</td>
<td>49 (44.55%)</td>
<td>49 (44.55%)</td>
<td>51 (46.36%)</td>
<td>16 (14.55%)</td>
<td>12 (10.91%)</td>
</tr>
<tr>
<td>&gt; 5%</td>
<td>19 (17.27%)</td>
<td>23 (20.90%)</td>
<td>23 (20.90%)</td>
<td>22 (20.00%)</td>
<td>18 (16.35%)</td>
<td>11 (10.00%)</td>
</tr>
<tr>
<td>&lt; -10%</td>
<td>13 (11.82%)</td>
<td>31 (28.19%)</td>
<td>30 (27.27%)</td>
<td>30 (27.27%)</td>
<td>67 (60.91%)</td>
<td>81 (73.64%)</td>
</tr>
<tr>
<td>-10% to 10%</td>
<td>92 (83.64%)</td>
<td>61 (55.45%)</td>
<td>62 (56.36%)</td>
<td>63 (57.27%)</td>
<td>32 (29.09%)</td>
<td>20 (18.18%)</td>
</tr>
<tr>
<td>&gt; 10%</td>
<td>5 (4.54%)</td>
<td>18 (16.36%)</td>
<td>18 (16.37%)</td>
<td>17 (15.45%)</td>
<td>11 (10.00%)</td>
<td>9 (8.18%)</td>
</tr>
<tr>
<td>Cohen’s $d$</td>
<td>0.0767</td>
<td>0.1513</td>
<td>0.1559</td>
<td>0.1342</td>
<td>0.9946</td>
<td>1.2022</td>
</tr>
</tbody>
</table>
Table 7. Efficiency of our approach in seconds on Siemens programs.

<table>
<thead>
<tr>
<th>Program</th>
<th>Instrumentation and control dependence path</th>
<th>SDPM</th>
<th>Fault localization</th>
</tr>
</thead>
<tbody>
<tr>
<td>print_token</td>
<td>8.335</td>
<td>526.9</td>
<td>0.524</td>
</tr>
<tr>
<td>print_token</td>
<td>6.248</td>
<td>113.1</td>
<td>0.315</td>
</tr>
<tr>
<td>replace</td>
<td>10.307</td>
<td>255.8</td>
<td>0.421</td>
</tr>
<tr>
<td>schedule</td>
<td>3.216</td>
<td>33.8</td>
<td>0.318</td>
</tr>
<tr>
<td>schedule2</td>
<td>0.204</td>
<td>101.8</td>
<td>0.412</td>
</tr>
<tr>
<td>tcas</td>
<td>0.035</td>
<td>1.156</td>
<td>0.009</td>
</tr>
<tr>
<td>tot-info</td>
<td>2.107</td>
<td>59.78</td>
<td>0.313</td>
</tr>
</tbody>
</table>
Table 8. Efficiency of our approach in seconds on UNIX programs.

<table>
<thead>
<tr>
<th>Program</th>
<th>Instrumentation and control dependence path</th>
<th>SDPM</th>
<th>Fault localization</th>
</tr>
</thead>
<tbody>
<tr>
<td>flex</td>
<td>122.4</td>
<td>5235</td>
<td>23.55</td>
</tr>
<tr>
<td>grep</td>
<td>114.8</td>
<td>4846</td>
<td>16.78</td>
</tr>
<tr>
<td>gzip</td>
<td>76.9</td>
<td>3887</td>
<td>11.24</td>
</tr>
<tr>
<td>sed</td>
<td>88.2</td>
<td>4165</td>
<td>12.27</td>
</tr>
</tbody>
</table>

TABLE 8 Efficiency of our approach in seconds on UNIX programs.
Highlights:

- We propose a state dependency probabilistic model and a fault-localization method.
- This model uses path profiles to capture more behavior state information.
- Our fault-localization approach analyzes the state dependency of program elements.