The Journal of Systems and Software 83 (2010) 188–208

Contents lists available at ScienceDirect

The Journal of Systems and Software

journal homepage: www.elsevier.com/locate/jss

A family of code coverage-based heuristics for effective fault localization ☆

W. Eric Wong [a,*], Vidroha Debroy [a], Byoungju Choi [b]

[a] Department of Computer Science, University of Texas at Dallas, TX 75083, USA
[b] Department of Computer Science and Engineering, Ewha womans University, Republic of Korea

ARTICLE INFO

Article history:
Received 3 March 2009
Received in revised form 27 July 2009
Accepted 22 September 2009
Available online 29 September 2009

Keywords:
Fault localization
Program debugging
Code coverage
Heuristics
Suspiciousness of code
Successful tests
Failed tests

ABSTRACT

Locating faults in a program can be very time-consuming and arduous, and therefore, there is an increased demand for automated techniques that can assist in the fault localization process. In this paper a code coverage-based method with a family of heuristics is proposed in order to prioritize suspicious code according to its likelihood of containing program bugs. Highly suspicious code (i.e., code that is more likely to contain a bug) should be examined before code that is relatively less suspicious; and in this manner programmers can identify and repair faulty code more efficiently and effectively. We also address two important issues: first, how can each additional failed test case aid in locating program faults; and second, how can each additional successful test case help in locating program faults. We propose that with respect to a piece of code, the contribution of the first failed test case that executes it in computing its likelihood of containing a bug is larger than or equal to that of the second failed test case that executes it, which in turn is larger than or equal to that of the third failed test case that executes it, and so on. This principle is also applied to the contribution provided by successful test cases that execute the piece of code. A tool, χDebug, was implemented to automate the computation of the suspiciousness of the code and the subsequent prioritization of suspicious code for locating program faults. To validate our method case studies were performed on six sets of programs: *Siemens suite, Unix suite, space, grep, gzip, and make*. Data collected from the studies are supportive of the above claim and also suggest Heuristics III(a), (b) and (c) of our method can effectively reduce the effort spent on fault localization.

1. Introduction

All software[1] must be tested and it must be tested well. However, when a test does reveal a fault[1] in the software, the burden is on the programmers to locate and fix the fault. The former task, fault localization, is recognized to be one of the most expensive and time-consuming debugging activities (Vessey, 1985). Furthermore, the larger and more complex code becomes, the longer it takes and the more difficult it is for programmers to manually locate faults without any help. One way to debug a program (specifically in the context of fault localization) when its observed behavior deviates from its expected behavior is to analyze its memory dump. Another way is to insert *print* statements around code that is thought to be suspicious such that the values of certain variables, which may provide hints towards the fault(s), are printed. The first approach suffers from inefficiency and intractability due to the tremendous amount of data that would need to be examined. The second approach places the burden on programmers to decide where to insert *print* statements as well as decide on which variable values to print. Clearly, such choices have a strong impact on the quality of the debugging as some choices are better and more meaningful than others. On the other hand, it is impractical to place a *print* statement at every location and display the value of every variable. Thus, this approach too is ineffective. A better strategy is to directly have the test cases lead the programmer to the location of the fault(s). This can be achieved by analyzing *how* a program is executed by the failed and the successful test cases in order to prioritize and rank the code based on its likelihood of containing bugs. Code with a higher priority should be examined first rather than that with a lower priority, as the former is more suspicious than the latter, i.e., more likely to contain the bugs.

Static slicing (Lyle and Weiser, 1987; Weiser, 1982) and dynamic slicing (Agrawal et al., 1996; Agrawal and Horgan, 1990; Korel and Laski, 1988) have been used to help programmers locate faults. More recently, execution slice-based fault localization methods have also been explored in several studies where an execution slice with respect to a given test case contains the set of

☆ This research was supported by the MKE (Ministry of Knowledge Economy), Korea, under the ITRC (Information Technology Research Center) support program supervised by the NIPA (National IT Industry Promotion Agency) (NIPA-2009-(C1090-0902-0032)).

* Corresponding author. Tel.: +1 972 883 6619; fax: +1 972 883 2399.
E-mail addresses: ewong@utdallas.edu (W. Eric Wong), vxd024000@utdallas.edu (V. Debroy), bjchoi@ewha.ac.kr (B. Choi).

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

0164-1212/$ - see front matter © 2009 Elsevier Inc. All rights reserved.
doi:10.1016/j.jss.2009.09.037

code executed by this test.[2] In Agrawal et al. (1995), a heuristic using an execution dice obtained by subtracting the execution slice of one successful test case from that of one failed test case is examined for its effectiveness in locating program faults. The tool χSlice, as part of the Telcordia's Visualization and Analysis Tool Suite (formerly χSuds), is developed to support this heuristic (Agrawal et al., 1998; χSuds User's manual, 1998). In Wong et al. (2005), more sophisticated heuristics using multiple successful and multiple failed test cases are applied to locate faults for software architectural design in SDL (a specification and description language (Ellsberger et al., 1997)). In Wong and Qi (2006), execution slice and inter-block data dependency-based heuristics are used to debug C programs. In Wong and Li (2005), we report how execution slice-based fault localization methods are used in an industrial setting for Java programs. Other fault localization methods have also been proposed including state-based (e.g., delta debugging (Zeller, 2002; Zeller and Hildebrandt, 2002), cause transition (Cleve and Zeller, 2005), and predicate switching (Zhang et al., 2006)), spectrum-based (such as Tarantula (Jones and Harrold, 2005) which uses the executable statement[3] coverage), statistical model-based (e.g., Liblit et al. (2005), SOBER (Liu et al., 2006) and Crosstab-based (Wong et al., 2008b)), and artificial neural network-based (e.g., BP-based (Wong and Qi, 2009) and RBF-based (Wong et al., 2008a)). However, none of these studies distinguishes the contribution of one failed test case from another or one successful test case from another. Thus, the objective of this paper is to address how the contribution of each additional test case (successful or failed) should be taken into account when computing the suspiciousness of code.

In order to do so, we propose a code coverage-based fault localization method. Given a piece of code (a statement in our case),[4] we first identify how many successful and failed tests execute it. We then explore whether all the failed test executions provide the same contribution towards program debugging, and whether all the successful test executions provide the same contribution. Our intuition tells us that the answer should be "no". Our proposal is that if a piece of code has already been executed successfully by 994 test cases, then the contribution of the 995th successful execution is likely to be less than that of the second successful execution when the code is only executed successfully once. In fact, for a given piece of code, the contribution of the first successful test case is larger than or equal to that of the second successful test case, which in turn is larger than or equal to that of the third successful test case, and so on. The same is also applied for failed test cases. Stated differently, for successful and failed test cases alike, we propose that the contribution of the kth successful (or failed) test case is always greater than or equal to that of the $(k+1)$th successful (or failed) test case. A family of heuristics is proposed each successively building on and addressing the deficiencies of previous heuristics. Each of them is unique in how it regulates the contribution of successful and failed test cases (see Section 2 for more details).

To demonstrate the feasibility of using our proposed method and to validate the correctness of our claims, case studies are performed on six different sets of programs: Siemens suite (Hutchins et al., 1994), Unix suite (Wong et al., 1998), space, grep, gzip, and make. These programs vary significantly in their size, function and the number of faults and test cases available. For the discussion in Section 2 and the studies reported in Section 3, each faulty version contains exactly one fault. The same approach has also been used by many other studies in fault localization (Cleve and Zeller, 2005; Jones and Harrold, 2005; Liblit et al., 2005; Liu et al., 2006; Renieris and Reiss, 2003; Zhang et al., 2006). However, in Section 4 we demonstrate the proposed method can easily be extended to handle programs with multiple faults as well. A tool (χDebug) based on this process is also implemented as it is not only time-consuming but also error-prone to manually prioritize code based on its likelihood of containing program bugs. Results from our study suggest that Heuristics III(a), (b), and (c) can perform better than other methods in locating program faults, as they lead to the examination of a smaller percentage of the code before the faults are located.

The rest of the paper is organized as follows. Section 2 describes our methodology. Case studies in which our method is applied to six different sets of programs are presented in Section 3. Also included is an introduction to our debugging tool χDebug. Section 4 explains how the method can be easily extended to handle programs with multiple bugs. Section 5 provides a discussion of certain relevant issues and in Section 6 we give an overview of related work. Finally in Section 7 we offer our conclusions and recommendations for future research.

2. Methodology

Given a piece of code (a statement, say $\mathscr{S}$, in our case) and a specific bug, say $\mathscr{B}$, suppose we want to determine how likely it is that $\mathscr{S}$ contains the bug $\mathscr{B}$. To facilitate the discussion in the rest of the paper, we define the following notations with respect to $\mathscr{S}$ and $\mathscr{B}$.

From Table 1, we have $\mathscr{N}_F \leqslant \Phi_F$, $\mathscr{N}_S \leqslant \Phi_S$, $\mathscr{N}_F = \sum_{i=1}^{\mathscr{G}_F} n_{F,i}$ and $\mathscr{N}_S = \sum_{i=1}^{\mathscr{G}_S} n_{S,i}$. Note that $\mathscr{N}_F$, $\mathscr{N}_S$, $n_{F,i}$, and $n_{S,i}$ depend on $\mathscr{S}$ (which statement is considered), whereas $\chi_{F/S}$ does not as its value is fixed for a given bug. Below we present three heuristics to show how the above information can be used to prioritize statements in terms of their likelihood of containing a program bug.

2.1. Heuristic I

If the program execution fails on a test case, it is natural to assume that, except for some special scenarios which will be discussed in Section 5, the corresponding bug resides in the set of statements executed by the failed test. In addition, if a statement is executed by two failed tests, our intuition suggests that this statement is more likely to contain the bug than a statement which is executed only by one failed test case. If we also assume every failed test case that executes $\mathscr{S}$ provides the same contribution

Table 1
Notations used in this paper.

Φ_F	Total number of failed test cases for $\mathscr{B}$
Φ_S	Total number of successful test cases for $\mathscr{B}$
$\mathscr{N}_F$	Total number of failed test cases with respect to $\mathscr{B}$ that execute $\mathscr{S}$
$\mathscr{N}_S$	Total number of successful test cases that execute $\mathscr{S}$
$c_{F,i}$	Contribution from the ith failed test case that executes $\mathscr{S}$
$c_{S,i}$	Contribution from the ith successful test case that executes $\mathscr{S}$
$\mathscr{G}_F$	Number of groups for the failed tests that execute $\mathscr{S}$
$\mathscr{G}_S$	Number of groups for the successful tests that execute S
$n_{F,i}$	Maximal number of failed test cases in the ith failed group
$n_{S,i}$	Maximal number of successful test cases in the ith successful group
$w_{F,i}$	Contribution from each test in the ith failed group
$w_{S,i}$	Contribution from each test in the ith successful group
$\chi_{F/S}$	Φ_F/Φ_S

[2] Given a program and a test case, an execution slice is the set of all the statements executed by that test, whereas the corresponding dynamic slice with respect to the output variables includes only those statements that are not only executed but also have an impact on the program output under that test. A more detailed comparison between static, dynamic, and execution slicing appears in Wong et al. (2005).

[3] All the comments, blank lines, and declarative statements such as function and variable declarations are excluded. When there is no ambiguity, we refer to "executable statements" as simply "statements" from this point on.

[4] For a fair comparison with other fault localization methods such as Tarantula (Jones and Harrold, 2005), the code coverage is measured in terms of "statements." However, it can also be measured in terms of other attributes such as blocks, decisions, c-uses and p-uses (Horgan and London, 1991).

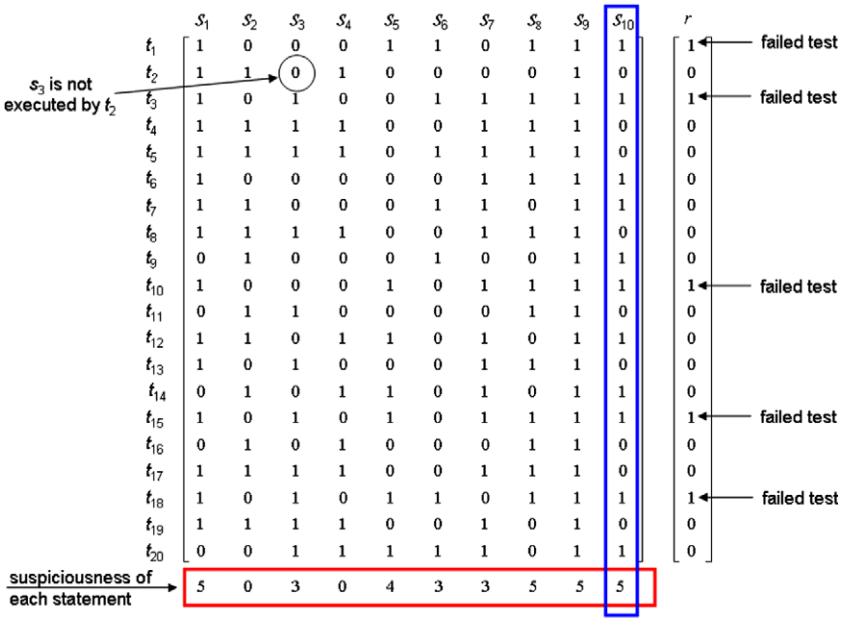


Fig. 1. An example of Heuristic I.

in program debugging (i.e., $c_{F,i} = c_{F,j}$ for $1 \leqslant i, j, \leqslant \mathscr{N}_F$), then we have the likelihood that $\mathscr{S}$ contains the bug is *proportional* to the number of failed tests that execute it. In this way, we define the suspiciousness of the jth statement as

$$\sum_{i=1}^{\mathscr{N}_F+\mathscr{N}_S} C_{i,j} \times r_i \text{ where } \begin{cases} C_{i,j} = 1, & \text{if the } i\text{th test case executes the } j\text{th statement} \\ C_{i,j} = 0, & \text{if the } i\text{th test case does not execute the } j\text{th statement} \\ r_i = 1, & \text{if the program execution on the } i\text{th test case fails} \\ r_i = 0, & \text{if the program execution on the } i\text{th test case succeeds} \end{cases} \quad (1)$$

Let us use the example in Fig. 1 to demonstrate how Heuristic I computes the suspiciousness of each statement. The left matrix shows how each statement is executed by each test case. An entry 1 indicates that the statement is executed by the corresponding test case and an entry 0 means it is not executed. For example, the entry at the intersection of t_2 and s_3 is 0 implies s_3 is not executed by t_2. The right matrix shows the execution result with an entry 1 for a failed execution and an entry 0 for a successful execution. Referring to Fig. 1, program execution fails on test cases t_1, t_3, t_{10}, t_{15}, and t_{18}. To compute the suspiciousness of $\mathscr{S}_{10}$, we find that it is executed by all five failed tests. From Eq. (1), the suspiciousness of $\mathscr{S}_{10}$ equals $\sum_{i=1}^{20}(C_{i,10} \times r_i) = 5$. Similarly, $\mathscr{S}_1$, $\mathscr{S}_8$ and $\mathscr{S}_9$ also have the same suspiciousness of 5 because each of them is executed by five failed tests, whereas $\mathscr{S}_5$ has a suspiciousness of 4 as it is executed by four failed tests, and $\mathscr{S}_3$, $\mathscr{S}_6$, and $\mathscr{S}_7$ have a suspiciousness of 3 for three failed executions. As for $\mathscr{S}_2$ and $\mathscr{S}_4$, their suspiciousness is zero because they are not executed by any failed test.

2.2. Heuristic II

In Heuristic I, we only take advantage of failed test cases to compute the suspiciousness of a statement. However, we also observe that if a statement is executed by a successful test case, its likelihood of containing a bug is reduced. Moreover, the more successful tests that execute a statement, the less likely that it contains a bug. If we also assume every successful test case that executes S provides the same contribution in program debugging (i.e., $c_{S,i} = c_{S,j}$ for $1 \leqslant i, j, \leqslant \mathscr{N}_S$), then we have the likelihood that $\mathscr{S}$ contains the bug is *inversely proportional* to the number of successful tests that execute it. Combined with our observation on the failed tests, we define the suspiciousness of the jth statement as

$$\sum_{i=1}^{\mathscr{N}_F+\mathscr{N}_S} C_{i,j} \times r_i - \sum_{i=1}^{\mathscr{N}_F+\mathscr{N}_S} C_{i,j} \times (1 - r_i) \text{ where } \begin{cases} C_{i,j} = 1, & \text{if the } i\text{th test case executes the } j\text{th statement} \\ C_{i,j} = 0, & \text{if the } i\text{th test case does not execute the } j\text{th statement} \\ r_i = 1, & \text{if the program execution on the } i\text{th test case fails} \\ r_i = 0, & \text{if the program execution on the } i\text{th test case succeeds} \end{cases} \quad (2)$$

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

Let us use the same matrices in Fig. 1 to demonstrate how Heuristic II computes the suspiciousness of each statement. From Eq. (2), the suspiciousness of $\mathscr{S}_{10}$ equals $\sum_{i=1}^{20} C_{i,10} \times r_i - \sum_{i=1}^{20} C_{i,10} \times$

$(1 - r_i) = 5 - 6 = -1$ .This is consistent with the fact that $\mathscr{S}_{10}$ is executed by five failed and six successful tests. Similarly, $\mathscr{S}_6$ also has the same suspiciousness of -1 because it is executed by three failed tests and four successful tests. Fig. 2 illustrates the suspiciousness computation based on Heuristic II.

2.3. Heuristic III

In the first two heuristics, we make no distinction between the contributions from different successful or failed test cases. The contribution provided by one failed test case is identical to that of each of the other failed test cases; similarly, a successful test case makes the same contribution as each of the other successful test cases. However, as explained in Section 1, if $\mathscr{S}$ has been executed by many successful test cases, then the contribution of each additional successful execution to the suspiciousness of $\mathscr{S}$ is likely to be less than that of the first few successful tests. Similarly, if $\mathscr{S}$ has already been executed by many failed test cases, the contribution of each additional failed execution to the suspiciousness of $\mathscr{S}$ is likely to be less than the contribution of the first few failed tests. Hence, we propose that for a given statement $\mathscr{S}$, the contribution introduced by the first successful test that executes it in computing its likelihood of containing a bug is larger than or equal to that of the second successful test that executes it, which is larger than or equal to that of the third successful test that executes it, and so on. This implies that $c_{S,1} \geq c_{S,2} \geq c_{S,3} \geq \ldots \geq c_{S,\mathscr{N}_S}$. The same also applies to failed tests, i.e., $c_{F,1} \geq c_{F,2} \geq c_{F,3} \geq \ldots \geq c_{F,\mathscr{N}_F}$.

One significant drawback of Heuristic II is that it cannot distinguish a statement (say $\mathscr{S}_\alpha$) executed by one successful and one failed test from another statement (say $\mathscr{S}_\beta$) executed by 10 successful and 10 failed tests. The suspiciousness of both $\mathscr{S}_\alpha$ and $\mathscr{S}_\beta$ computed by using Eq. (2) is zero as the first one is $1 - 1 = 0$ and the second one is $10 - 10$ which is also zero. This is counter-intuitive because $\mathscr{S}_\beta$ should be more suspicious than $\mathscr{S}_\alpha$, as the former is executed by more failed tests than the latter.

To overcome this problem, we propose that if the statement $\mathscr{S}$ is executed by at least one failed test, then the total contribution from all the successful tests that execute $\mathscr{S}$ should be less than the total contribution from all the failed tests that execute $\mathscr{S}$ (namely, $\sum_{i=1}^{\mathscr{N}_S} c_{S,i} < \sum_{k=1}^{\mathscr{N}_F} c_{F,k}$.) We assume failed tests that execute $\mathscr{S}$ are divided into $\mathscr{G}_F$ groups such that the first failed group has at most $n_{F,1}$ tests (the first $n_{F,1}$ tests), the second failed group has at most $n_{F,2}$ tests from the remaining tests, the third has at most $n_{F,3}$ tests from the rest, and so on. These groups are filled in order such that the test cases are assigned to each group starting from the first to the last group. We propose that all the tests in the same failed group have the same contribution towards program debugging, but tests from different groups have different contributions. For example, every test in the ith failed group has a contribution of $w_{F,i}$ and every test in the jth failed group ($i \neq j$) has a contribution of $w_{F,j}$ which is different from $w_{F,i}$. The same also applies to the successful tests that execute $\mathscr{S}$. With this in mind, the suspiciousness of $\mathscr{S}$ can then be defined as

$$\sum_{i=1}^{\mathscr{G}_F} w_{F,i} \times n_{F,i} - \sum_{i=1}^{\mathscr{G}_S} w_{S,i} \times n_{S,i} \text{ and } \sum_{i=1}^{\mathscr{N}_S} c_{S,i} < \sum_{k=1}^{\mathscr{N}_F} c_{F,k} \quad (3)$$

Let us use the same matrices in Fig. 1 to demonstrate how Heuristic III computes the suspiciousness of each statement. For illustrative proposes, we set $\mathscr{G}_F = \mathscr{G}_S = 3$, $n_{F,1} = n_{S,1} = 2$, and $n_{F,2} = n_{S,2} = 4$. That is, the first failed (or successful) group has at most two tests, the second group has at most four from the remaining, and the third has everything else, if any. We also assume each test case in the first, second, and third failed groups gives a contribution of 1, 0.1 and 0.01, respectively ($w_{F,1} = 1$, $w_{F,2} = 0.1$, and $w_{F,3} = 0.01$). Similarly, we set $w_{S,1} = 1$, $w_{S,2} = 0.1$, and $w_{S,3}$ to be a small value defined as $\alpha \times \chi_{F/S}$ where α is a scaling factor (please refer to Section 5.1 for a discussion on α). Eq. (3) can be rewritten as

$$[(1.0 \times n_{F,1} + (0.1) \times n_{F,2} + (0.01) \times n_{F,3}] - [(1.0) \times n_{S,1} + (0.1) \times n_{S,2} + \alpha \times \chi_{F/S} \times n_{S,3}] \quad (4)$$

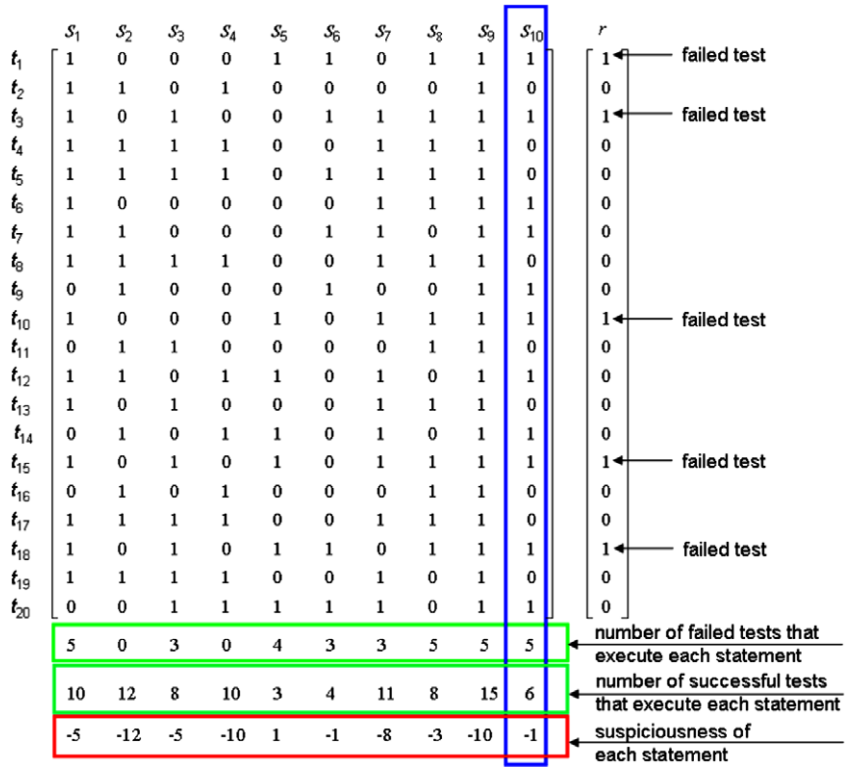
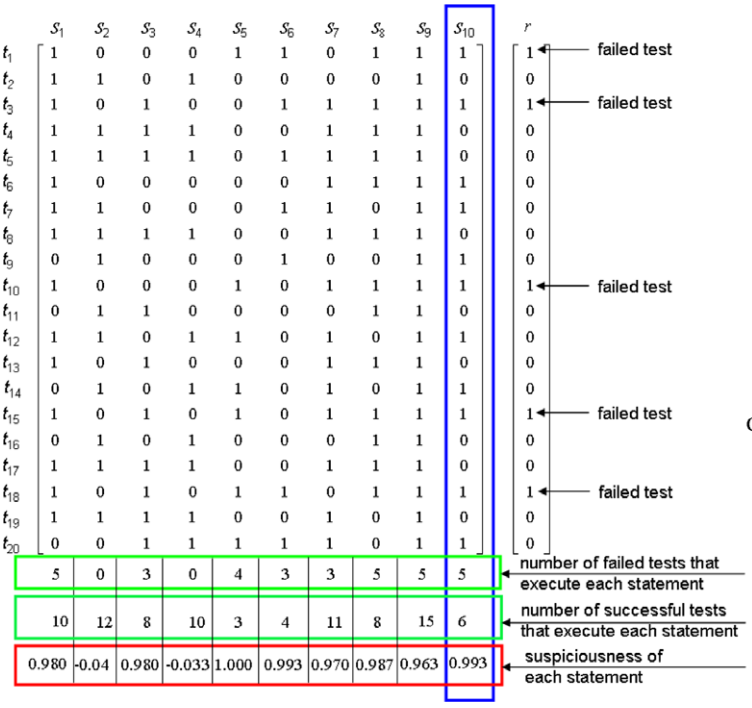


Fig. 2. An example of Heuristic II.



	$n_{F,1}$	$n_{F,2}$	$n_{F,3}$	$\sum_{i=1}^{\mathcal{N}_F} c_{F,i}$	$n_{S,1}$	$n_{S,2}$	$n_{S,3}$	$\sum_{i=1}^{\mathcal{N}_S} c_{S,i}$
$\mathcal{S}_1$	2	3	0	2.3	1	3	6	1.320
$\mathcal{S}_2$	0	0	0	0	0	0	12	0.040
$\mathcal{S}_3$	2	1	0	2.1	1	1	6	1.120
$\mathcal{S}_4$	0	0	0	0	0	0	10	0.033
$\mathcal{S}_5$	2	2	0	2.2	1	2	0	1.200
$\mathcal{S}_6$	2	1	0	2.1	1	1	2	1.107
$\mathcal{S}_7$	2	1	0	2.1	1	1	9	1.130
$\mathcal{S}_8$	2	3	0	2.3	1	3	4	1.313
$\mathcal{S}_9$	2	3	0	2.3	1	3	11	1.337
$\mathcal{S}_{10}$	2	3	0	2.3	1	3	2	1.307

Computation based on the number of test cases in the first, second, and third failed and successful groups

Fig. 3. An example of Heuristic III.

$$\text{where } n_{F,1} = \begin{cases} 0, & \text{for } \mathcal{N}_F = 0 \\ 1, & \text{for } \mathcal{N}_F = 1 \\ 2, & \text{for } \mathcal{N}_F \geq 2 \end{cases}$$

$$n_{F,2} = \begin{cases} 0, & \text{for } \mathcal{N}_F \leq 2 \\ \mathcal{N}_F - 2, & \text{for } 3 \leq \mathcal{N}_F \leq 6 \\ 4, & \text{for } \mathcal{N}_F > 6 \end{cases}$$

$$n_{F,3} = \begin{cases} 0, & \text{for } \mathcal{N}_F \leq 6 \\ \mathcal{N}_F - 6, & \text{for } \mathcal{N}F > 6 \end{cases} \text{ and}$$

$$n_{S,1} = \begin{cases} 0, & \text{for } n_{F,1} = 0, 1 \\ 1, & \text{for } n_{F,1} = 2 \text{ and } \mathcal{N}_S \geq 1 \end{cases}$$

$$n_{S,2} = \begin{cases} 0, & \text{for } \mathcal{N}_S \leq n_{S,1} \\ \mathcal{N}_S - n_{S,1}, & \text{for } n_{S,1} < \mathcal{N}_S < n_{F,2} + n_{S,1} \\ n_{F,2}, & \text{for } \mathcal{N}_S \geq n_{F,2} + n_{S,1} \end{cases}$$

$$n_{S,3} = \begin{cases} 0, & \text{for } \mathcal{N}_S < n_{S,1} + n_{S,2} \\ \mathcal{N}_S - n_{S,1} - n_{S,2}, & \text{for } \mathcal{N}_S \geq n_{S,1} + n_{S,2} \end{cases}$$

From Eq. (4), when $\alpha = 0.01$, the suspiciousness of $\mathcal{S}_{10}$ equals

$$[(1.0) \times 2 + (0.1) \times 3 + (0.01) \times 0] - [(1.0) \times 1 + (0.1) \times 3 + 0.01 \times 5/15 \times 2] = 2.3 - 1.3 - 0.02/3 = 0.993$$

and the suspiciousness of $\mathcal{S}_2$ equals $[(1.0) \times 0 + (0.1) \times 0 + (0.01) \times 0] - [(1.0) \times 0 + (0.1) \times 0 + 0.01 \times 5/15 \times 12] = -0.040$. Fig. 3 gives the suspiciousness computation based on Heuristic III. We observe the following:

- Neither $\mathcal{S}_2$ nor $\mathcal{S}_4$ is executed by any failed test case, but $\mathcal{S}_2$ is executed by 12 successful tests and $\mathcal{S}_4$ is only executed by 10 successful tests. The suspiciousness of $\mathcal{S}_2$ computed by Eq. (4) is -0.040; that is less than -0.033, the suspiciousness of $\mathcal{S}_4$. Similarly, $\mathcal{S}_{10}$, $\mathcal{S}_8$, $\mathcal{S}_1$ and $\mathcal{S}_9$ are all executed by five failed tests, but by different numbers of successful tests (namely, 6, 8, 10 and 15, respectively). Their suspiciousness computed by Eq. (4) is 0.993, 0.987, 0.980 and 0.963. Both cases are consistent with our claim that "more successful executions imply less likely to contain the bug."
- All the statements, except $\mathcal{S}_2$ and $\mathcal{S}_4$, have at least one failed test case and their suspiciousness computed by using Eq. (4) satisfies $\sum_{i=1}^{\mathcal{N}_S} c_{S,i} < \sum_{k=1}^{\mathcal{N}_F} c_{F,k}$. This is also consistent with the requirement set by Heuristic III.

Note that in the case of Heuristic III, when we state that the contribution of the kth successful (or failed) test case is larger than or equal to that of the $(k+1)$th successful (or failed) test case, we do not refer to any test case in particular. Our suspiciousness computation is independent of the way in which the test cases have been ordered. For explanatory purposes, let us assume that test cases are independent of each other,[5] this implies whether one test case is executed before or after another does not change the coverage and execution result in any way. Thus, the total number of successful tests and the total number of failed tests are fixed regardless of the execution order. This also implies that for the same grouping mechanism (e.g., the first failed (or successful) group has at most two tests (the first two), the second group has at most four from the remaining, and the third has everything else, if any), the number of test cases assigned to each failed group and each successful group is fixed regardless of the execution order. Thus, our heuristic does not rely on the order of test case execution and shall be applicable regardless of execution plan.

3. Case studies

In this section we report our case studies using three Heuristics (I, II and III) for effective fault localization across six different sets of programs. We first provide an overview of the programs, test cases and bugs used in our study. A debugging tool, χDebug, is explained subsequently. We then present an analysis of our results and a comparison of our methods with Tarantula (Jones and Harrold, 2005) which has been shown to be more effective than other fault localization methods such as set-union, set intersection, near-

[5] Refer to the discussion on "Bugs due to a sequence of test cases" in Section 5.2.

est-neighbor (Renieris and Reiss, 2003) and cause transition techniques (Cleve and Zeller, 2005).

3.1. Programs, test cases, and bugs

3.1.1. Siemens suite

Several fault localization related studies (Cleve and Zeller, 2005; Jones and Harrold, 2005; Renieris and Reiss, 2003) have employed the programs from the Siemens suite in their experiments. The correct versions, 132 faulty versions of the programs and all the test cases are downloaded from The Siemens Suite (2007). Of these 132 faulty versions, three have been excluded from our study: version 9 of "schedule2" because all the tests pass on this version and therefore there is no failed test; versions 4 and 6 of "print_tokens" because the faults are in the header files instead of the C files and we do not possess the coverage information with respect to header files. A previous fault localization study (Jones and Harrold, 2005) utilizing the Siemens suite excludes 10 faulty versions. Among those excluded are versions 27 and 32 of "replace" and versions 5, 6, and 9 of "schedule" because the tool used (gcc with gcov) does not collect the coverage information properly in the event of program crash due to a segmentation fault. We are able to make use of these faulty versions due to a revised version of χSuds (χSuds User's manual, 1998) which can collect and record runtime trace information in spite of the above problem. We also use all of the test cases downloaded from the web site which is constantly updated. Therefore, the number of tests used in this study is slightly larger than that reported in Jones and Harrold (2005). Version 10 of "print_tokens" and version 32 of "replace" are excluded from the study in Jones and Harrold (2005) as none of the test cases fails on them. However, as a consequence of using a larger test set we are able to record failures on these versions and include them in our experiments. Multi-line statements are combined as one source code line, so that they are counted as one executable statement, just as in Jones and Harrold (2005).

A summary of each program including the name and a brief description, the number of faulty versions, LOC (the size of the program before any non-executable code is removed), number of executable statements and number of test cases is presented in Table 2.

3.1.2. Unix suite

The Unix suite consists of 10 Unix utility programs. The faulty versions used are created by the application of mutation-based fault injection, which has been shown in a recent study (Andrews et al., 2005) to be an effective approach to simulating realistic faults that can be used in software testing research. The same set of programs and test cases were also used in other studies (Wong et al., 1998; Wong and Qi, 2009) but the number of faulty versions used in this study differs from that reported in Wong et al. (1998) because the execution environments are different and because this study makes use of a revised version of χSuds as described in User's manual (1998). Table 3 gives a summary of this suite. More descriptions of the test case generation, fault set, and erroneous program preparation can be found in Wong et al. (1998).

3.1.3. space

Jones and Harrold (2005) reported that the space program has 6218 lines of executable code. However, following the same convention as described in Jones and Harrold (2005), if we combine

Table 2
Summary of the Siemens suite.

Program	Description	Number of faulty versions	LOC	Number of executable statements	Number of test cases
print_tokens	Lexical analyzer	7	565	175	4130
print_tokens2	Lexical analyzer	10	510	178	4115
Replace	Pattern replacement	32	563	216	5542
Schedule	Priority scheduler	9	412	121	2650
Schedule2	Priority scheduler	10	307	112	2710
tcas	Altitude separation	41	173	55	1608
tot_info	Information measure	23	406	113	1052

Table 3
Summary of the Unix suite.

Program	Description	Number of faulty versions	LOC	Number of executable statements	Number of test cases
Cal	Print a calendar for a specified year or month	20	202	88	162
Checkeq	Report missing or unbalanced delimiters and .EQ/.EN pairs	20	102	57	166
Col	Filter reverse paper motions from nroff output for display on a terminal	30	308	165	156
Comm	Select or reject lines common to two sorted files	12	167	76	186
Crypt	Encrypt and decrypt a file using a user supplied password	14	134	77	156
Look	Find words in the system dictionary or lines in a sorted list	14	170	70	193
Sort	Sort and merge files	21	913	448	997
Spline	Interpolate smooth curves based on given data	13	338	126	700
Tr	Translate characters	11	137	81	870
Uniq	Report or remove adjacent duplicate lines	17	143	71	431

Table 4
Summary of space, gzip, grep and make.

Program	Description	Number of faulty versions	LOC	No. of executable statements	Number of test cases
space	Provide a user interface to configure an array of antennas	38	9126	3657	13585
grep	Search for a pattern in a file	19	12653	3306	470
gzip	Reduce the size of named files using the Lempel–Ziv coding	28	6573	1670	211
make	Manage building of executables and other products from code	31	20014	5318	793

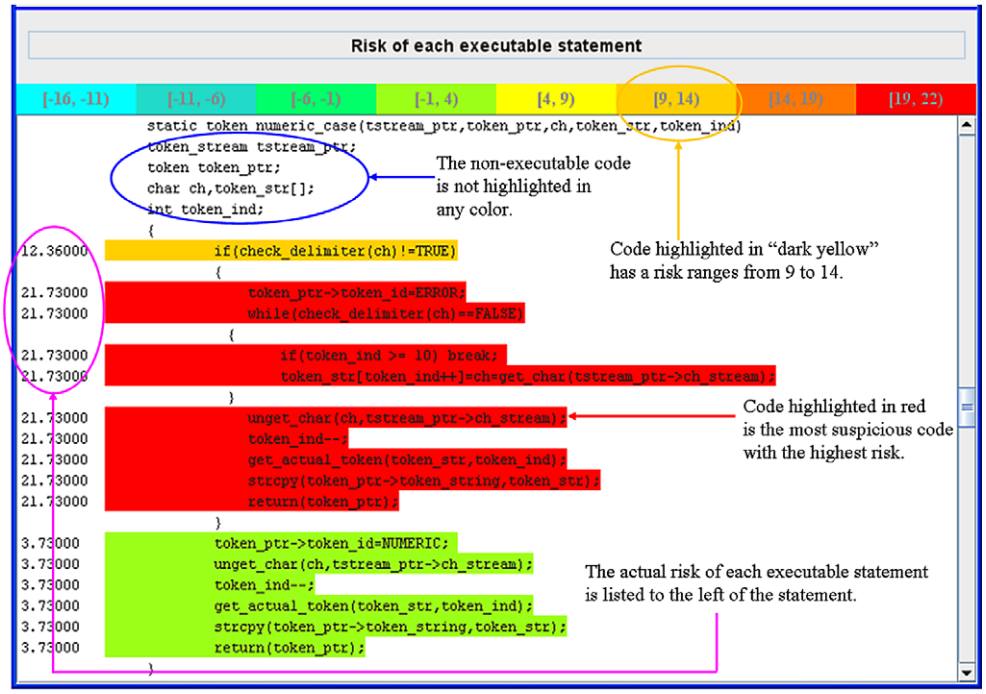


Fig. 4. Visualizing suspicious code for possible bug locations.

multi-line statements as one source code line we have 3657 executable statements. The correct version, the 38 faulty versions, and a suite of 13585 test cases used in this study are downloaded from http://sir.unl.edu/portal/index.html. Three faulty versions are not used in our study because none of the test cases fails on these faulty versions whereas eight faulty versions are excluded in Jones and Harrold (2005) for various reasons.

3.1.4. grep

The source code of version 2.2 of the grep program is downloaded from http://sir.unl.edu/portal/index.html, together with a suite of 470 test cases and 18 bugs. Compared with the study in Liu et al. (2006), where none of these bugs can be detected by any test case in the suite, four bugs are detected in our environment. Two additional bugs injected by Liu et al. (2006) are also used. The authors of Liu et al. (2006) argue that although faults are manually injected, they do mimic realistic logic errors. We follow a similar approach to inject 13 additional bugs. With the addition of these faults, altogether, there are 19 faulty versions.

3.1.5. gzip

Version 1.1.2 of the gzip program with 16 seeded bugs and 217 test cases is also downloaded from http://sir.unl.edu/portal/index.html. Nine faults are excluded from the study since none of the test cases fails on them and six test cases are discarded because they cannot be executed in our environment. Once again we follow a similar approach as described in Andrews et al. (2005) and Liu et al. (2006) to inject 21 bugs in addition to the 7 usable original bugs. In total, 28 faulty versions were used.

3.1.6. make

Version 3.76.1 of make is downloaded from http://sir.unl.edu/portal/index.html, together with 793 test cases and 19 faulty versions of the program. Of these, 15 faulty versions are excluded as they contain bugs which cannot be detected by any of the downloaded test cases in our environment. Using the above fault injection approach, we generate an additional 27 bugs for a total of 31 usable faulty versions.

Table 4 gives a summary of the space, grep, gzip and make programs. A list of all the additional bugs for grep, gzip and make is available upon request.

Each of the programs used in our studies varies from the other in terms of size, functionality, number of faulty versions and number of test cases. This allows us to better generalize the findings and results of this paper. Currently, each faulty version contains exactly one bug which may span multiple statements or even multiple functions. However, we emphasize that the method proposed in this paper can be easily extended to handle programs with multiple bugs as well, and this is further discussed in Section 4.

3.2. χDebug: a debugging tool based on code coverage

Tool support would allow us to automate the fault localization process as well as the suspiciousness computation of each statement and the highlighting of suspicious statement(s). Recognizing this, we developed χDebug: a debugging tool that supports the method presented in Section 2.

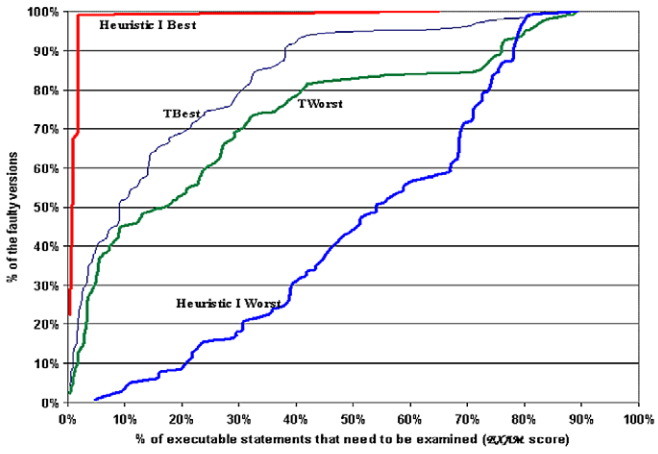
Given a program and its successful and failed test cases, χDebug displays the source code and its corresponding suspiciousness in a user-friendly GUI as shown in Fig. 4.[6] Each executable statement is highlighted in an appropriate color with its suspiciousness listed on the left. The numerical range associated with each color in the spectrum at the top gives the suspiciousness range represented by that color. The most suspicious code is highlighted in red, and the non-executable statements are not highlighted in any color. Depending on the program under examination, different numbers of colors (up to seven) with different numerical ranges associated with each color are automatically decided upon by χDebug.

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

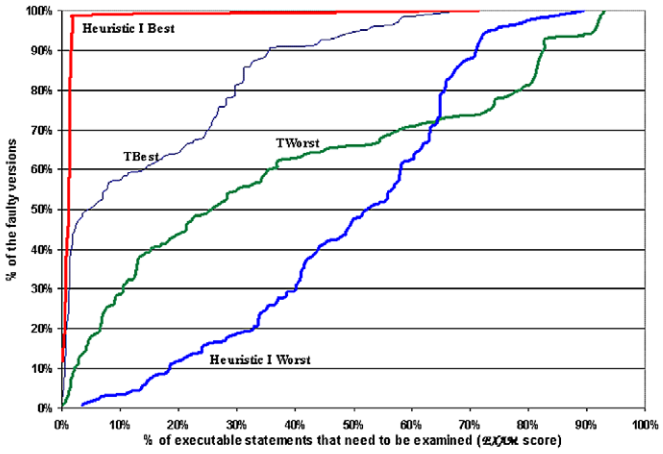
In addition to the graphical user interface allowing programmers to visualize suspicious code for possible bug locations, χDebug also provides a command-line text interface via shell scripts for a controlled experiment like the one reported here. In the latter, the bug locations are already known before the experiment, and the objective is not to *find* the location of a bug but to *verify* whether the bug is in the suspicious code identified by using our method. Appropriate routines can be invoked directly to compute the suspiciousness of each statement using the heuristics discussed in Section 2 and check whether such suspicious code indeed contains the bug(s). The percentage of code that has to be examined before a given bug is detected (namely, the $\mathscr{EXAM}$ score defined in Section 3.3) is also computed. It is this approach which makes χDebug very robust for our case study.

3.3. Data collection

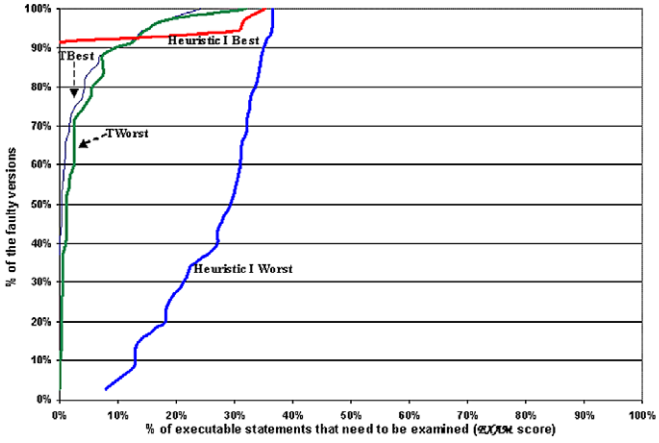
For the Siemens suite, Unix suite and space, all executions were on a PC with a 2.13 GHz Intel Core 2 Duo CPU and 8 GB physical memory. The operating system was SunOS 5.10 (Solaris 10) and



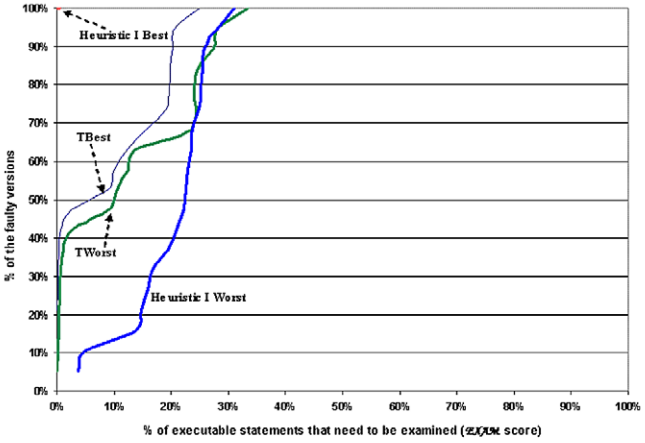
Part (a). Effectiveness comparison on the Siemens suite



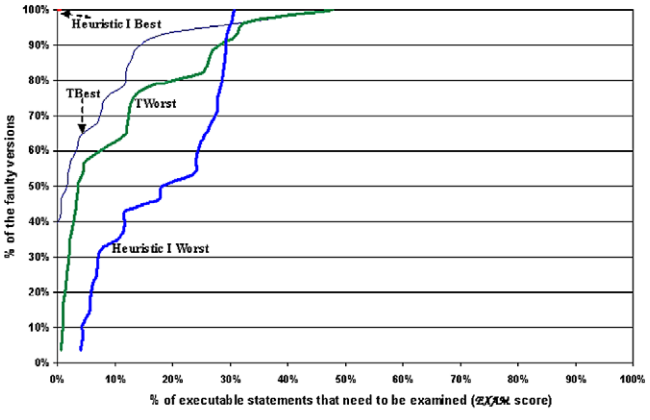
Part (b). Effectiveness comparison on the Unix suite



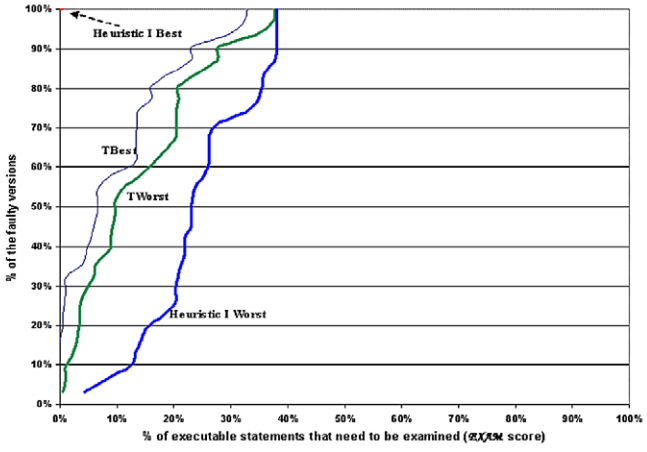
Part (c). Effectiveness comparison on the space program



Part (d). Effectiveness comparison on the grep program



Part (e). Effectiveness comparison on the gzip program



Part (f). Effectiveness comparison on the make program

Fig. 5. Effectiveness comparison between the Heuristic I method and the Tarantula method.

the compiler used was gcc 3.4.3. For grep, gzip and make, the executions were on a Sun-Fire-280R machine with SunOS 5.10 as the operating system and gcc 3.4.4 as the compiler. Each faulty version was executed against all its corresponding available test cases. The success or failure of a test case execution was determined by comparing the outputs of the faulty version and the correct version of a program. During a test case execution χSuds has the ability to record which statements of the source code are executed by a test case and how many times each of the statements is executed. In order for a statement to have been covered by a test case we require that it must have been executed by the test case at least once. If not, then the statement is not covered by the test case. The revised version of χSuds (χSuds User's manual, 1998) can collect runtime trace correctly even if a program execution was crashed due to a segmentation fault. In our study, all the comments, blank lines, declarative statements (e.g., function and variable declarations) are excluded for analysis.

It is essential to point out that our focus is on finding a good starting point from which programmers can begin to fix a bug, rather than to provide the complete set of code that would need to be corrected with respect to each bug. With this in mind, even though a bug may span multiple statements, which may not be contiguous, the fault localization process ceases when the first statement corresponding to the bug is identified.

In Section 3.4, results of our method are compared with those of Tarantula (Jones and Harrold, 2005) in order to evaluate the relative effectiveness of each method. For a fair comparison, we compute the effectiveness of Tarantula using our test data and their ranking mechanism. Note that statistics such as fault revealing behavior and statement coverage of each test can vary under different compilers, operating systems, and hardware platforms. The ability of the coverage measurement tool (revised χSuds versus gcc with gcov) to properly handle "segmentation faults" also has an impact on the use of certain faulty versions. We performed cross-checks against the original reported data on the Siemens suite and the space program whenever possible in order to verify the correctness of our computations. The other programs reported in this paper had not also been studied by the authors of Jones and Harrold (2005) and Jones et al. (2007).

Previous studies (Jones and Harrold, 2005; Renieris and Reiss, 2003) assigned a score to every faulty version of each subject program, which was defined as the percentage of the program that need not be examined to find a faulty statement in the program. In this paper, we measure the effectiveness of a fault localization method by a score $\mathscr{EXAM}$ which describes the percentage of statements that need to be examined until the first bug-containing statement is reached. Both of the two scores provide similar information (in fact one score can easily be derived from the other), but we find that the $\mathscr{EXAM}$ score is more direct and easier to understand. Liu et al. (2006) used a score to represent the percentage of code a developer needs to examine before the fault location is found. However, it is different from $\mathscr{EXAM}$ score in that their score is computed based on a program dependence graph, while the $\mathscr{EXAM}$ score is computed based on the ranking of all statements, although the two are comparable. The effectiveness of different fault localization methods can be compared based on $\mathscr{EXAM}$. For a faulty version Ω, if its $\mathscr{EXAM}$ score assigned by method A is less than that assigned by method B (that is, method A can guide the programmer to the fault in Ω by examining less code than method B), then A is said to be more effective than B for locating the fault in Ω. If there is more than one faulty version, then A is more effective than B if A assigns a smaller $\mathscr{EXAM}$ score to a greater number of faulty versions than B.

We note that the same suspiciousness value may be assigned to multiple statements. Let us assume a bug-containing statement and several correct statements both share the same suspiciousness. Then, in the best case we examine the bug-containing statement first and in the worst case we examine it last and have to examine

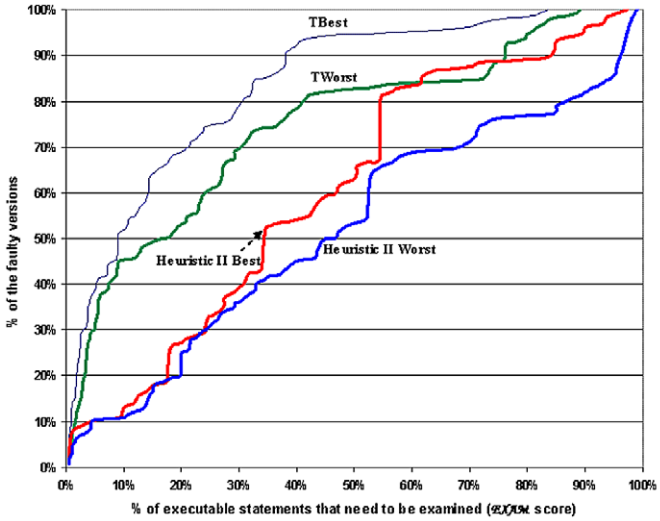
Table 5
Pair-wise comparison between Heuristic I and Tarantula.

		Heuristic I Best versus TBest			Heuristic I Worst versus TWorst			Heuristic I Worst versus TBest		
Siemens	More effective	110			27			10		
	Same effectiveness		18			1			2	
	Less effective			1			101			117
Unix	More effective	101			67			17		
	Same effectiveness		70			2			0	
	Less effective			1			103			155
space	More effective	21			1			1		
	Same effectiveness		11			0			0	
	Less effective			3			34			34
grep	More effective	15			8			2		
	Same effectiveness		4			0			0	
	Less effective			0			11			17
gzip	More effective	17			3			2		
	Same effectiveness		11			0			0	
	Less effective			0			25			26
make	More effective	26			9			6		
	Same effectiveness		5			0			0	
	Less effective			0			22			25

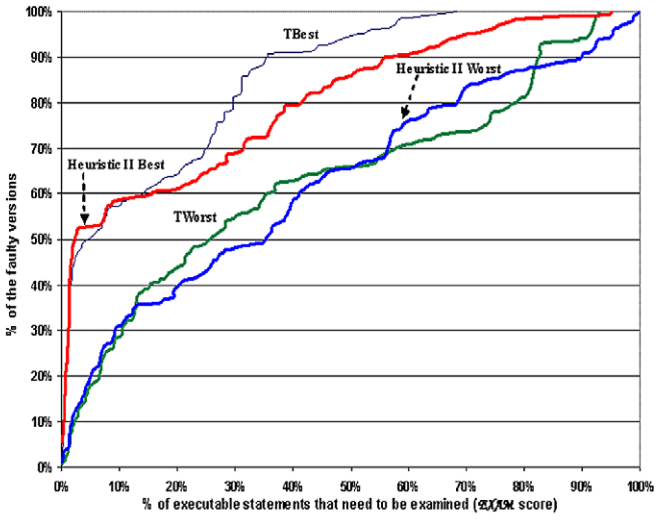
Table 6
Total number of statements examined by Heuristic I and Tarantula.

Program	Heuristic I Best	TBest	Heuristic I Best/TBest (%)	Heuristic I Worst	TWorst	Heuristic I Worst/TWorst (%)
Siemens	208	2453	8.48	7519	3311	227.09
Unix	2462	3364	73.19	6030	7629	79.04
space	3566	3876	92	33764	5094	662.82
grep	19	5793	0.33	12728	7812	162.93
gzip	28	3110	0.9	8506	5032	169.04
make	31	16890	0.18	40860	23468	174.11

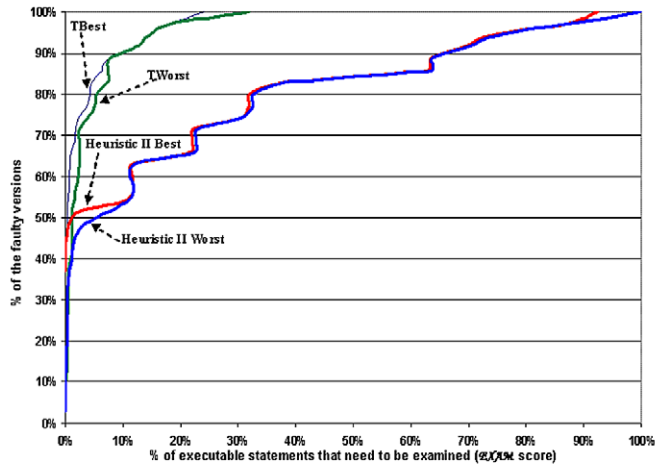
many correct statements before we discover the bug. This results in two different levels of effectiveness – the "best" and the "worst". In all our experiments we assume that for the "best" effectiveness we examine the faulty statement first and for the "worst" effectiveness we examine the faulty statement last. Furthermore, data are presented for each of these two levels of effectiveness. Hereafter, we refer to the best effectiveness of the Tarantula method as TBest and the worst effectiveness as TWorst. We follow this naming con-



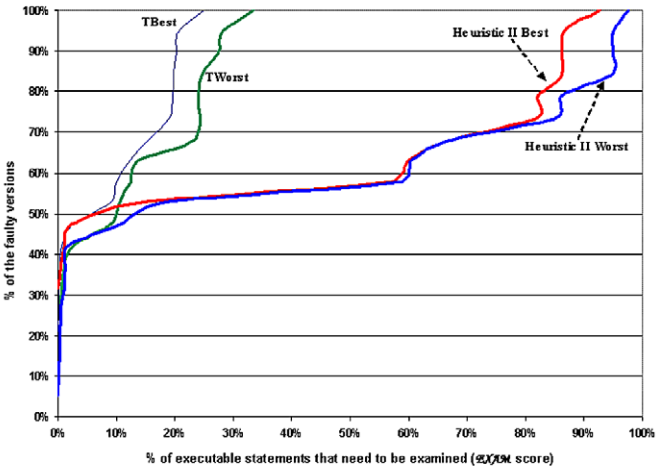
Part (a). Effectiveness comparison on the Siemens suite



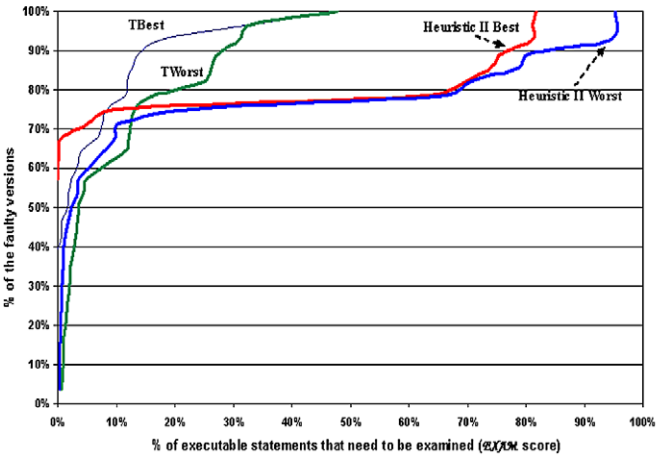
Part (b). Effectiveness comparison on the Unix suite



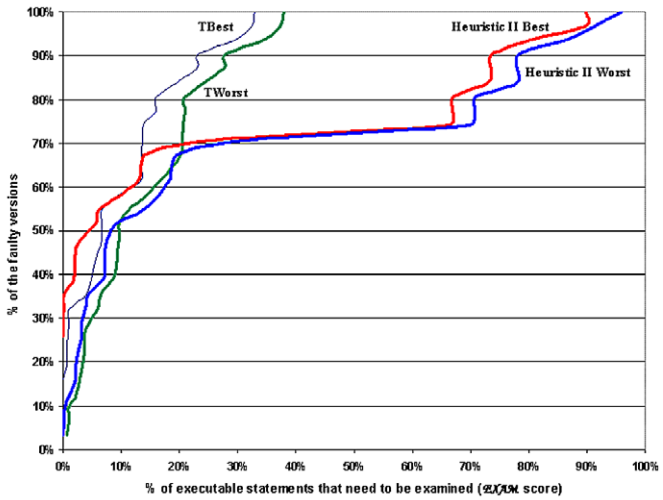
Part (c). Effectiveness comparison on the space program



Part (d). Effectiveness comparison on the grep program



Part (e). Effectiveness comparison on the gzip program



Part (f). Effectiveness comparison on the make program

Fig. 6. Effectiveness comparison between the Heuristic II method and the Tarantula method.

vention for each of the heuristics applied to our method (Heuristic I Best, Heuristic I Worst, and so on) and maintain this style for all of the data presented.

3.4. Results and analysis

From Jones and Harrold (2005), we observe that for the seven programs in the Siemens suite, the Tarantula method is more effective in fault localization than the set-union, set intersection, nearest-neighbor, and cause-transitions methods (Cleve and Zeller, 2005; Renieris and Reiss, 2003). Refer to Section 6 for a description of these methods. To avoid having too many curves in each figure which reduces its readability significantly, we only focus on the comparison between the effectiveness of the Tarantula method and that of our method. If we can show that our method is more effective in fault localization than the Tarantula method, then our method is also more effective than the set-union, set intersection, nearest-neighbor, and cause-transitions methods.

3.4.1. Heuristic I versus Tarantula

The comparison between the effectiveness of Heuristic I and Tarantula is shown in Fig. 5. The two curves labeled as TBest and TWorst give the best and the worst effectiveness of the Tarantula method. The two labeled as "Heuristic I Best" and "Heuristic I Worst" give the best and the worst effectiveness of our method using Heuristic I. In the case of grep, gzip and make only one statement needs to be examined per fault for "Heuristic I Best" which explains the shape of the curves corresponding to these programs in Parts (d), (e) and (f). We observe that the best effectiveness of Heuristic I is in general better than the best effectiveness of Tarantula, but its worst effectiveness is in general worse than the worst effectiveness of Tarantula. We also note that the gap between the best and the worst curves of Heuristic I is too large. The actual effectiveness of Heuristic I can be any value between "Heuristic I Best" and "Heuristic I Worst" which is very unstable. Therefore, Heuristic I by itself is not a very good strategy for localizing program bugs.

Data are also provided in Table 5 to show the pair-wise comparison between the effectiveness of the Heuristic I method and the Tarantula method to decide for how many faulty versions our method outperforms another, equals another, and underperforms another. For example, for the Unix suite, Heuristic I Best is more effective (examining fewer statements before detecting the first faulty statement) than TBest for 101 of the 172 faulty versions; as effective as TBest (examining the same number of statements) for 70 versions; and is less effective than TBest (examining more statements) for one fault. Comparisons are also presented between Heuristic I Worst and TWorst as well as between Heuristic I Worst and TBest.

Table 6 presents the effectiveness comparison in terms of the total number of statements that need to be examined to locate all the faults. Corresponding to the observations from Fig. 5 and Table 5, we also note that Heuristic I Best is able to detect the faults for each program by examining a fewer number of statements than TBest. However, Heuristic I Worst is only able to detect the faults by examining more statements than TWorst for all of the programs except the Unix suite. As an example, in the case of grep, Heuristic I Best is able to detect the faults by examining just 19 statements whereas TBest does so by examining 5793 statements. In contrast Heuristic I Worst detects the faults in grep by examining 12728 statements whereas TWorst does so in 7812 statements.

Heuristic I only makes use of the failed test cases while computing the suspiciousness of statements. Based on the data we observe that Heuristic I is unreliable by itself when applied to fault localization. We now present data on the same sets of programs based on Heuristic II, which makes use of not just failed test cases but also the successful ones.

Table 7
Pair-wise comparison between Heuristic II and Tarantula.

		Heuristic II Best versus TBest			Heuristic II Worst versus TWorst			Heuristic II Worst versus TBest		
Siemens	More effective	1			1			0		
	Same effectiveness		10			10			5	
	Less effective			118			118			124
Unix	More effective	40			77			19		
	Same effectiveness		72			35			10	
	Less effective			60			60			143
space	More effective	8			12			5		
	Same effectiveness		10			7			1	
	Less effective			17			16			29
grep	More effective	6			6			5		
	Same effectiveness		4			4			1	
	Less effective			9			9			13
gzip	More effective	10			15			5		
	Same effectiveness		11			6			1	
	Less effective			7			7			22
make	More effective	17			22			9		
	Same effectiveness		5			0			0	
	Less effective			9			9			22

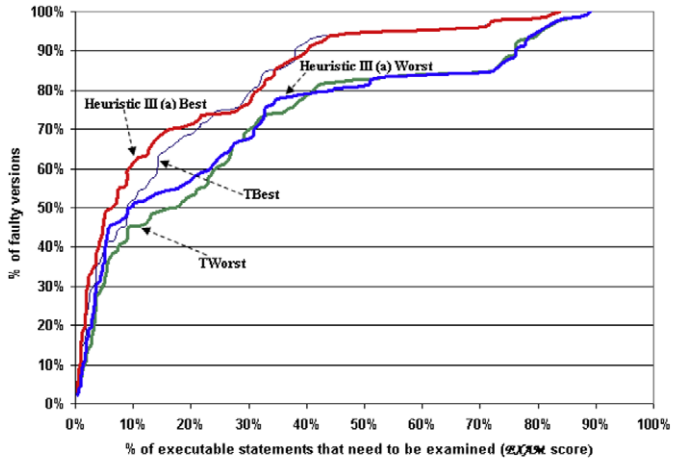
Table 8
Total number of statements examined by Heuristic II and Tarantula.

Program	Heuristic II Best	TBest	Heuristic II Best/TBest (%)	Heuristic II Worst	TWorst	Heuristic II Worst/TWorst (%)
Siemens	6335	2453	258.26	7187	3311	217.06
Unix	5504	3364	163.61	8934	7629	117.11
space	25469	3876	657.09	26567	5094	521.54
grep	23806	5793	410.94	25606	7812	327.78
gzip	9087	3110	292.19	10968	5032	217.97
make	41734	16890	247.09	48401	23468	206.24

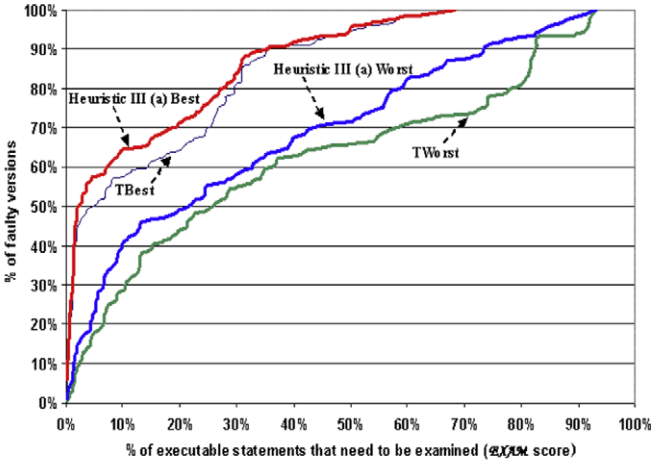
3.4.2. Heuristic II versus Tarantula

Fig. 6 gives the comparison between Heuristic II and Tarantula. All the legends and naming conventions are the same as in Fig. 5. We observe that the gap between "Heuristic II Best" and "Heuristic II Worst" is much smaller when compared with the gap in Fig. 5.

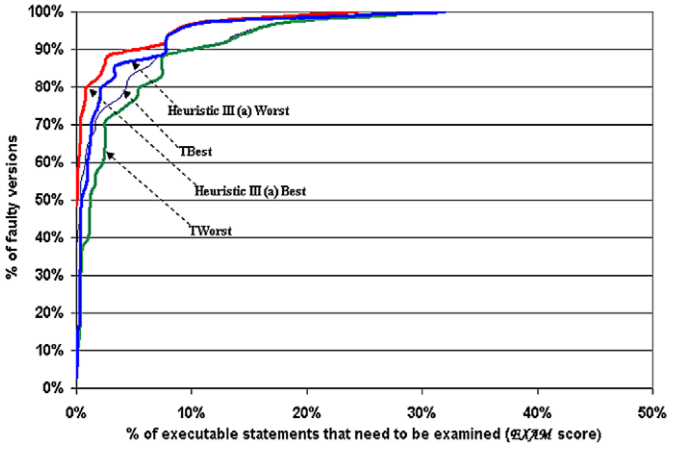
Data are also provided in Table 7, in a manner similar to that of Table 5 to show the pair-wise comparison between the effectiveness of the Heuristic II method and the Tarantula method. When comparing Heuristic II Best to TBest, we observe that in general the former is less effective than the latter for the majority of the programs, the two exceptions being gzip and make. In the case of the comparison between Heuristic II Worst and TWorst, we observe that Heuristic II Worst is better than TWorst for the Unix suite, gzip and make; and that TWorst is better for the Siemens suite, space and grep.



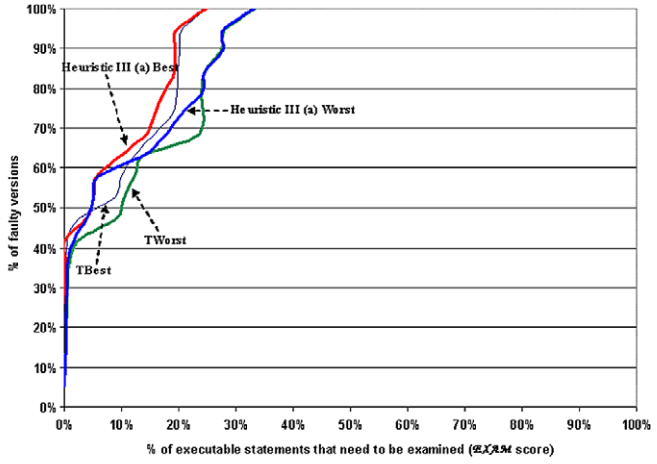
Part (a). Effectiveness comparison on the Siemens suite



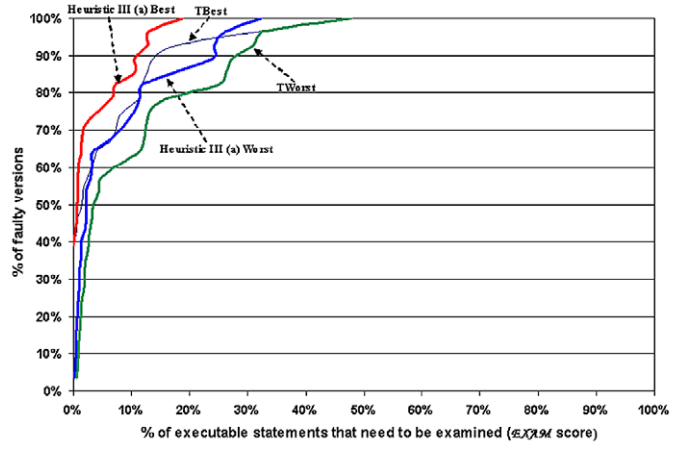
Part (b). Effectiveness comparison on the Unix suite



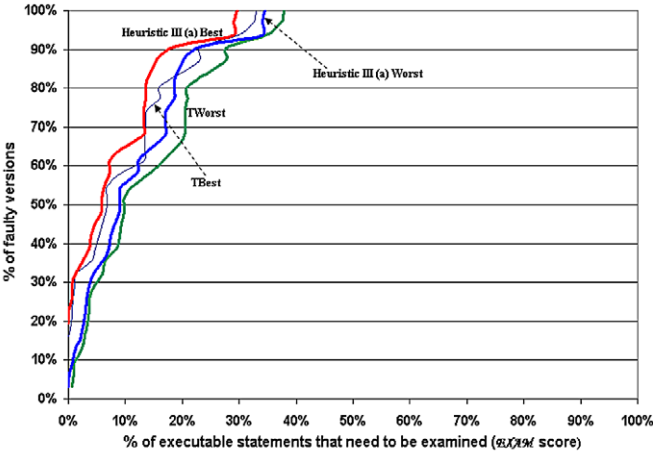
Part (c). Effectiveness comparison on the space program



Part (d). Effectiveness comparison on the grep program



Part (e). Effectiveness comparison on the gzip program



Part (f). Effectiveness comparison on the make program

Fig. 7. Effectiveness comparison between the Heuristic III(a) method and the Tarantula method.

Table 8, similar to Table 6, presents the effectiveness comparison between Heuristic II and Tarantula in terms of the total number of statements that need to be examined to locate all the faults. We find that for all of the programs, irrespective of whether the best or worst cases are considered, Heuristic II performs worse than Tarantula in terms of the number of statements that need to be examined for the faults to be discovered.

The data above is strongly suggestive that Heuristic II as is cannot be used for the purposes of fault localization. Heuristic II should intuitively perform better than Heuristic I because the former also makes use of successful test cases (more information) than the latter. However, this is not the case in our studies. The reason is that while Heuristic I only considers failed test cases; Heuristic II considers failed and successful test cases, but does not calibrate the contribution provided by failed and successful test cases in any way. Heuristic II simply computes the suspiciousness of a statement as the difference between the number of failed test cases that execute it and the number of successful test cases that execute it (refer to Eq. (2) in Section 2). In such a scenario, even if a statement is executed by some failed tests, it is still possible to have a negative suspiciousness if the number of successful tests that execute the statement is larger than the failed. Recall in Section 2, to address this issue we propose that if a statement is executed by at least one failed test, then the total contribution from all the successful tests that execute the statement should be less than that of the failed. Heuristic II has this deficiency and this paves the way for Heuristic III.

3.4.3. Heuristic III

For Heuristic III, we emphasize that $c_{F,1} \geq c_{F,2} \geq c_{F,3} \geq \ldots \geq c_{F,\mathscr{N}_F}$, $c_{S,1} \geq c_{S,2} \geq c_{S,3} \geq \ldots \geq c_{S,\mathscr{N}_S}$ and $\sum_{i=1}^{\mathscr{N}_S} c_{S,i} < \sum_{k=1}^{\mathscr{N}_F} c_{F,k}$ (see Table 1 for their definitions). Eq. (3) gives the generic formula for this heuristic. However, prior to its application we need to assign values to certain parameters such as $\mathscr{G}_F$, $\mathscr{G}_S$, $w_{F,i}$ and $w_{S,i}$. In our studies, we choose the same values as the ones used in the example in Section 2. This means that the formula for Heuristic III used here is the same as the one in Eq. (4) which is an instance of Eq. (3). To ensure that $\sum_{i=1}^{\mathscr{N}_S} c_{S,i} < \sum_{k=1}^{\mathscr{N}_F} c_{F,k}$, the contribution of each test in the third successful group (i.e., $w_{S,3}$) has to be regulated. There are different ways to accomplish this. One way is to make $w_{S,3} = \alpha \times \chi_{F/S}$ where α is a scaling factor and $\chi_{F/S}$ is the ratio of the number of failed tests and successful tests for a given bug. A discussion on the impact of α and $\chi_{F/S}$ on the effectiveness of fault localization appears in Section 5.1. We perform our experiments using three values of α: 0.01, 0.001 and 0.0001. The results obtained from Heuristic III using these three values are presented below and are respectively labeled as Heuristic III(a), Heuristic III(b) and Heuristic III(c).

3.4.4. Heuristic III(a) versus Tarantula with $\alpha = 0.01$

Fig. 7 shows the graphs obtained for Heuristic III(a) across the six sets of programs in a fashion similar to that of Figs. 5 and 6. We observe that in general the curves corresponding to Heuristic III(a) Best and Heuristic III(a) Worst are either above or close to those of TBest and TWorst, respectively. This shows considerable improvement over Heuristics I and II.

Table 9, as before, shows the pair-wise comparison between Heuristic III(a) and Tarantula. Based on this comparison we observe that for each of the programs under study, irrespective of whether the best or worst effectiveness is considered, Heuristic III(a) consistently outperforms Tarantula.

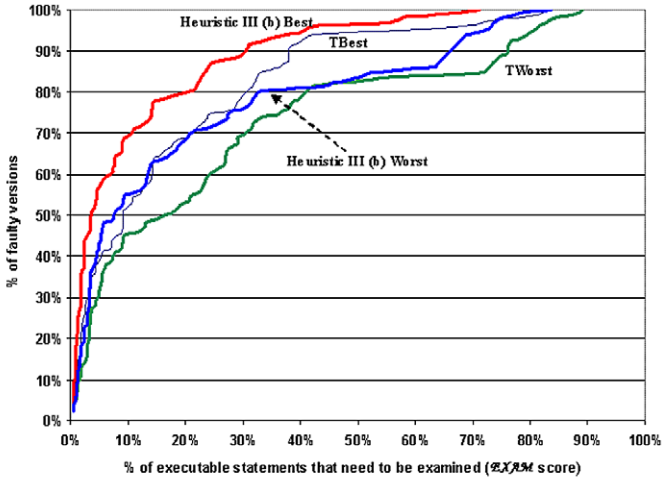
Table 10 compares Heuristic III(a) and Tarantula in terms of the total number of statements that must be examined in order for all of the faults to be detected. We observe that irrespective of whether the best or worst effectiveness is considered, Heuristic III(a) outperforms Tarantula on every program by examining fewer statements. For the space program, the ratio of the number of

Table 9
Pair-wise comparison between Heuristic III(a) and Tarantula.

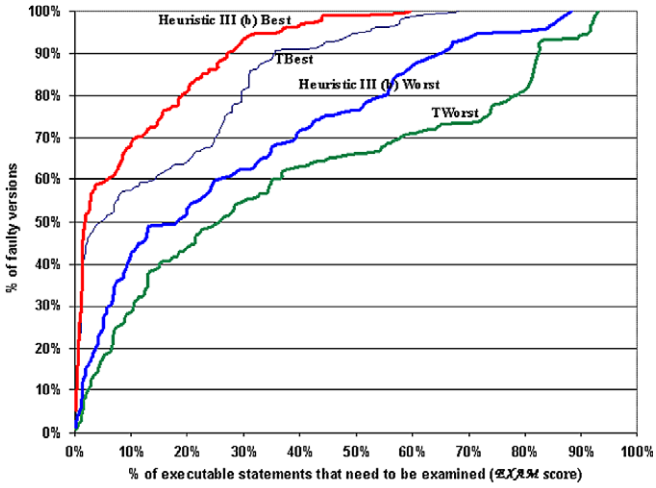
		Heuristic III(a) Best versus TBest			Heuristic III(a) Worst versus TWorst			Heuristic III(a) Worst versus TBest		
Siemens	More effective	43			43			25		
	Same effectiveness		70			70			14	
	Less effective			16			16			90
Unix	More effective	57			101			24		
	Same effectiveness		107			63			15	
	Less effective			8			8			133
space	More effective	17			21			13		
	Same effectiveness		14			11			1	
	Less effective			4			3			21
grep	More effective	8			9			5		
	Same effectiveness		7			6			1	
	Less effective			4			4			13
gzip	More effective	13			18			5		
	Same effectiveness		13			8			0	
	Less effective			2			2			23
make	More effective	23			28			7		
	Same effectiveness		7			2			0	
	Less effective			1			1			24

Table 10
Total number of statements examined by Heuristic III(a) and Tarantula.

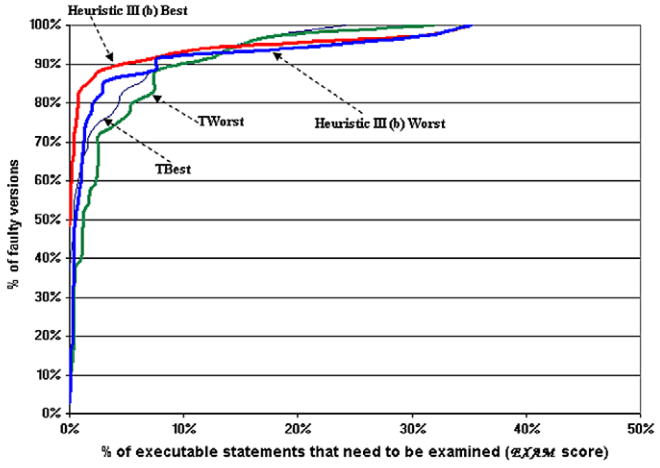
Program	Heuristic III(a) Best	TBest	Heuristic III(a) Best/TBest (%)	Heuristic III(a) Worst	TWorst	Heuristic III(a) Worst/TWorst (%)
Siemens	2177	2453	88.75	3035	3311	91.66
Unix	2804	3364	83.35	6175	7629	80.94
space	2287	3876	59	3364	5094	66.04
grep	5170	5793	89.25	6903	7812	88.36
gzip	1584	3110	50.93	3361	5032	66.79
make	14045	16890	83.16	19784	23468	84.30



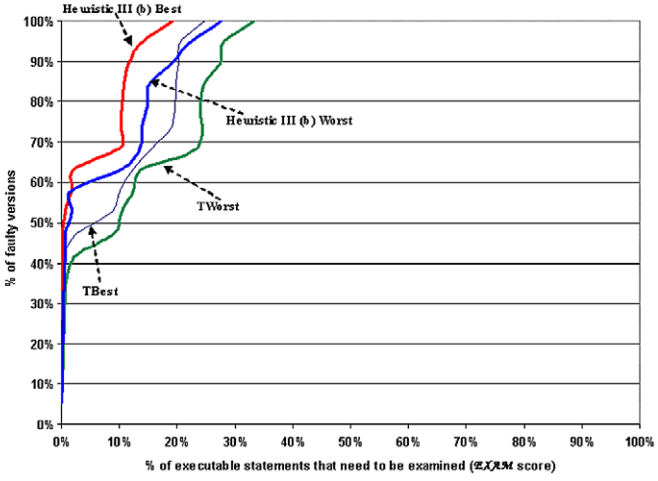
Part (a). Effectiveness comparison on the Siemens suite



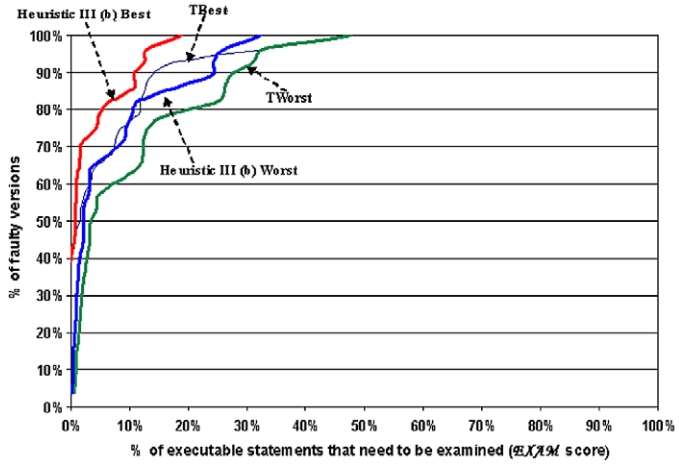
Part (b). Effectiveness comparison on the Unix suite



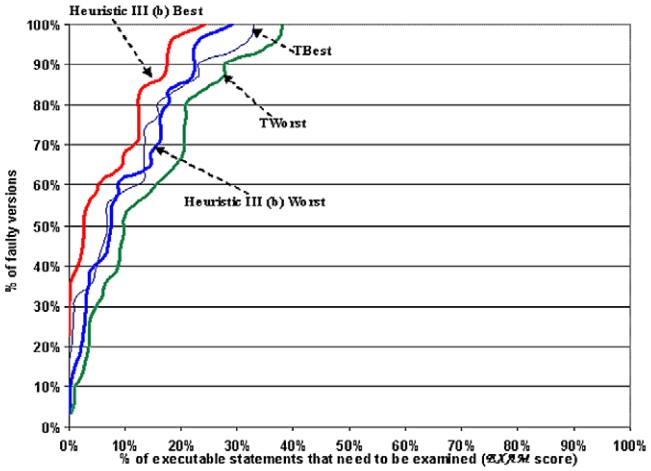
Part (c). Effectiveness comparison on the space program



Part (d). Effectiveness comparison on the grep program



Part (e). Effectiveness comparison on the gzip program



Part (f). Effectiveness comparison on the make program

Fig. 8. Effectiveness comparison between the Heuristic III(b) method and the Tarantula method.

statements examined by Heuristic III(a) Best for locating all the faults to that by Tarantula is just 0.59. Note that this does not imply the set of statements examined by Heuristic III(a) is a subset of those examined by Tarantula.

Clearly the performance of our method significantly improves as a result of the calibration applied to the contributions of the failed and successful tests. To better understand the role of α, we now continue the experiment with $\alpha = 0.001$.

Table 11
Pair-wise comparison between Heuristic III(b) and Tarantula.

		Heuristic III(b) Best versus TBest			Heuristic III(b) Worst versus TWorst			Heuristic III(b) Worst versus TBest		
Siemens	More effective	66			65			43		
	Same effectiveness		57			57			12	
	Less effective			6			7			74
Unix	More effective	80			124			38		
	Same effectiveness		89			45			13	
	Less effective			3			3			121
space	More effective	18			22			14		
	Same effectiveness		14			11			1	
	Less effective			3			2			20
grep	More effective	11			12			10		
	Same effectiveness		7			6			1	
	Less effective			1			1			8
gzip	More effective	14			19			5		
	Same effectiveness		12			7			0	
	Less effective			2			2			23
make	More effective	23			28			14		
	Same effectiveness		7			2			0	
	Less effective			1			1			17

Table 12
Total number of statements examined by Heuristic III(b) and Tarantula.

Program	Heuristic III(b) Best	TBest	Heuristic III(b) Best/TBest (%)	Heuristic III(b) Worst	TWorst	Heuristic III(b) Worst/TWorst (%)
Siemens	1439	2453	58.66	2335	3311	70.52
Unix	1701	3364	50.56	5072	7629	66.48
space	3503	3876	90.38	4580	5094	89.91
grep	3019	5793	52.11	4752	7812	60.83
gzip	1535	3110	49.36	3313	5032	65.84
make	10817	16890	64.04	16556	23468	70.55

3.4.5. Heuristic III(b) versus Tarantula with $\alpha = 0.001$

Heuristic III(b) continues the application of Heuristic III when the value of the parameter α is 0.001. The curves presented in Fig. 8 present the effectiveness curves for Heuristic III(b) and Tarantula using the same conventions as previous figures have done. As with Heuristic III(a) we observe that for both the best and worst cases, the curve for Heuristic III(b) is either above or close to that of Tarantula for all of the programs under study.

Table 11 depicts the pair-wise comparison between Heuristic III(b) and Tarantula. As with Heuristic III(a) we observe that Heuristic III(b) outperforms Tarantula irrespective of whether the best or worst effectiveness is considered across any of the programs in our studies. In fact for the grep program, we note that not only Heuristic III(b) Best but also III(b) Worst are more effective than TBest for a greater number of faults. Furthermore, the effectiveness of Heuristic III(b) is always better than or equal to that of Heuristic III(a) for both the best and the worst cases based on a comparison of Tables 11 and 10.

Table 12 shows the comparison between Heuristic III(b) and Tarantula in terms of the total number of statements examined by either method to detect the faults in each program. As with Heuristic III(a) we note that irrespective of whether the best or worst effectiveness is considered, Heuristic III(b) outperforms Tarantula in terms of the number of statements examined. Additionally, when compared to Heuristic III(a), with the exception of the space program, Heuristic III(b) always examines a fewer number of statements than Heuristic III(a) to detect the faults.

The reduction of α from 0.01 to 0.001 provides better results in general. To establish whether this trend holds true for further reductions in α, we now decrease the value of α to 0.0001.

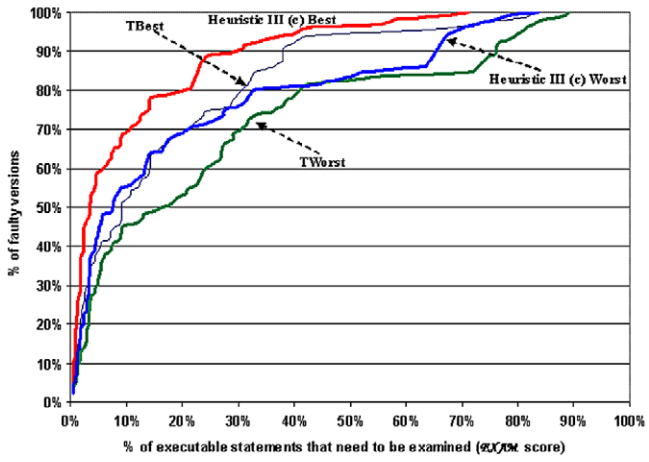
3.4.6. Heuristic III(c) versus Tarantula with $\alpha = 0.0001$

Fig. 9 displays the effectiveness comparison between Heuristic III(c) and Tarantula. The curves for both the best and worst effectiveness are provided following the same conventions as before. We observe that in the case of Heuristic III(c), not only is the best effectiveness better than that of TBest, but in several cases such as grep and make, the worst effectiveness of Heuristic III(c) is comparable to that of TBest.

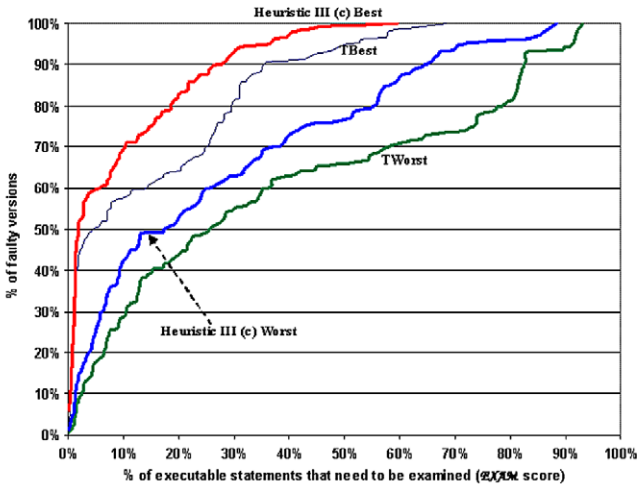
Table 13 depicts the pair-wise comparison between Heuristic III(c) and Tarantula. As with Heuristics III(a) and III(b), we observe that Heuristic III(c) yields better performance than Tarantula independent of whether we consider the best or the worst effectiveness for all of the programs under study. Based on a comparison of Tables 13, 11 and 9, we find that the improvement in effectiveness between Heuristics III(c) and III(b) is not as great as the improvement between Heuristics III(b) and III(a). However, the effectiveness of Heuristic III(c) is always better than or equal to that of Heuristic III(b).

Table 14 compares Heuristic III(c) and Tarantula in terms of the total number of statements that must be examined to detect the faults in each program. We find as with Heuristics III(a) and (b), that Heuristic III(c) Best is always better than TBest and that Heuristic III(c) Worst is always better than TWorst. In fact for three out of the six programs studied, the number of statements examined by Heuristic III(c) Best for detecting all of the faults is less than half of the number of statements examined by TBest. Note once again that the set of statements examined by Heuristic III may not be a subset of those examined by Tarantula.

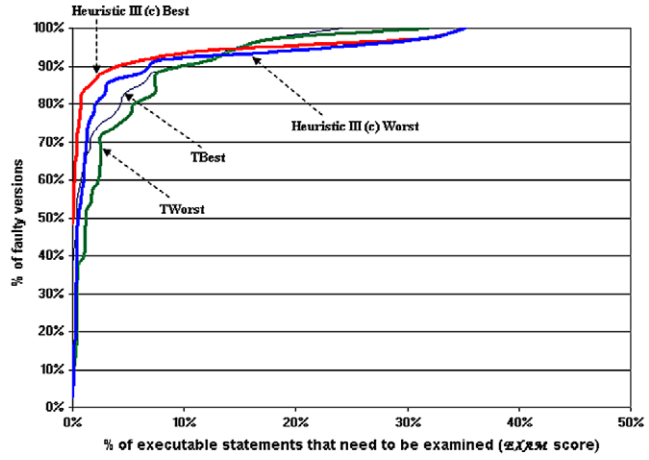
For the purposes of evaluating Heuristic III, three values of α have been considered and the corresponding data have been presented and analyzed. The data suggest that smaller values of α provide an increase in effectiveness. However, the improvement



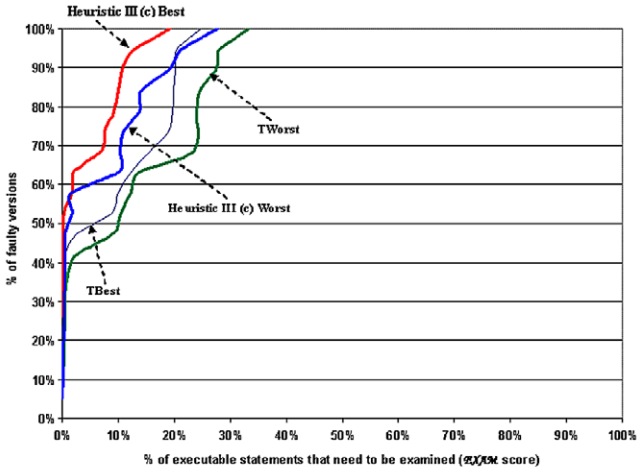
Part (a). Effectiveness comparison on the Siemens suite



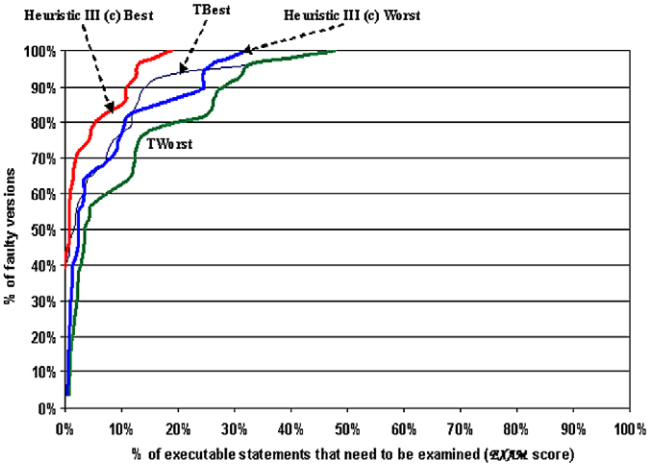
Part (b). Effectiveness comparison on the Unix suite



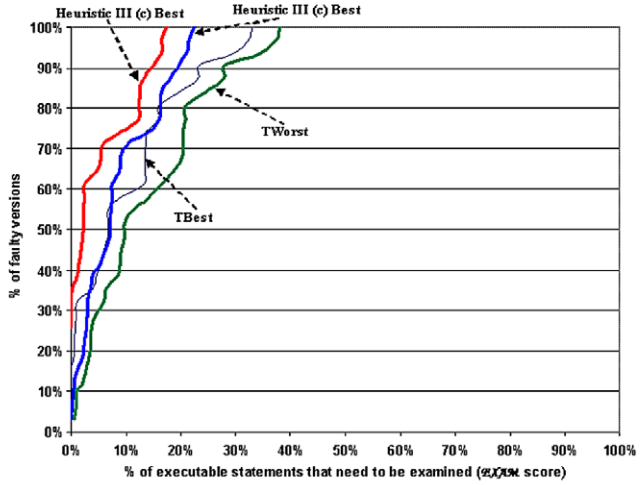
Part (c). Effectiveness comparison on the space program



Part (d). Effectiveness comparison on the grep program



Part (e). Effectiveness comparison on the gzip program



Part (f). Effectiveness comparison on the make program

Fig. 9. Effectiveness comparison between the Heuristic III(c) method and the Tarantula method.

in effectiveness between III(b) and III(a) seems to be larger than that of III(c) and III(b) which is indicative that the gain obtained by a reduction of α suffers from saturation and decreases along with the value of α. A more detailed discussion appears in Section 5.1.

4. Programs with multiple bugs

The case studies discussed so far are on programs which only contain one bug per faulty version. However, the method proposed in this paper can also be extended to perform fault localization on

Table 13
Pair-wise comparison between Heuristic III(c) and Tarantula.

		Heuristic III(c) Best versus TBest			Heuristic III(c) Worst versus TWorst			Heuristic III(c) Worst versus TBest		
Siemens	More effective	68			67			43		
	Same effectiveness		55			55			13	
	Less effective			6			7			73
Unix	More effective	80			124			38		
	Same effectiveness		89			45			13	
	Less effective			3			3			121
space	More effective	18			22			14		
	Same effectiveness		14			11			1	
	Less effective			3			2			20
grep	More effective	11			12			10		
	Same effectiveness		7			6			1	
	Less effective			1			1			8
gzip	More effective	14			19			5		
	Same effectiveness		12			7			0	
	Less effective			2			2			23
make	More effective	23			28			16		
	Same effectiveness		7			2			0	
	Less effective			1			1			15

Table 14
Total number of statements examined by Heuristic III(c) and Tarantula.

Program	Heuristic III(c) Best	TBest	Heuristic III(c) Best/TBest (%)	Heuristic III(c) Worst	TWorst	Heuristic III(c) Worst/TWorst (%)
Siemens	1396	2453	56.91	2292	3311	69.22
Unix	1655	3364	49.2	5026	7629	65.88
space	3478	3876	89.73	4555	5094	89.42
grep	2702	5793	46.64	4435	7812	56.77
gzip	1535	3110	49.36	3312	5032	65.82
make	8533	16890	50.52	14272	23468	60.81

programs with multiple bugs as well. This can be achieved by first grouping failed test cases into fault-focused clusters such that those in the same cluster are related to the same fault (Jones et al., 2007). Then, the failed tests in each cluster can be combined with the successful tests in order to locate the fault that cluster is related to. This also means that we can apply the method proposed in this paper to locate these faults and therefore our method can be extended to handle programs with multiple bugs as well. Additional clustering techniques can also be found in literature. For example, Zheng et al. (2006) performed a clustering on failing executions based on fault-predicting predicates. Liu and Han used two distance measures to cluster failed test cases (Liu and Han, 2006).

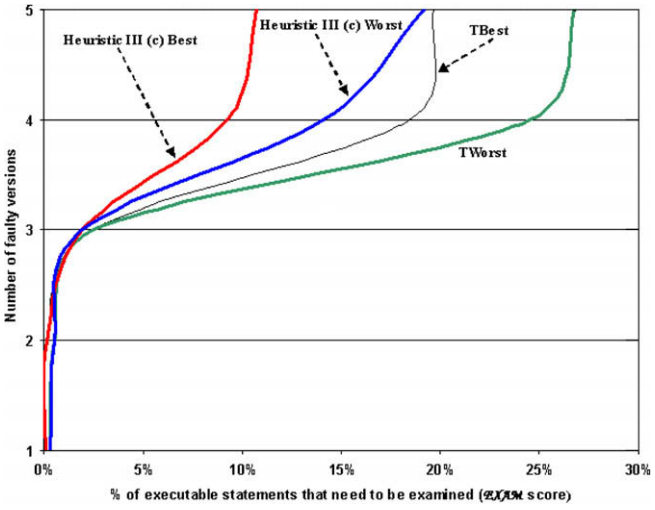


Fig. 10. Effectiveness comparison between the Heuristic III(c) method and the Tarantula method.

Podgurski et al. (2003) analyzed the execution profiles of failed test cases and correlate each of them with program bugs. The result is a clustering of failed executions based on the underlying bug responsible for the failure.

For the purposes of demonstrating the application of our method on programs with multiple bugs, we perform a case study on a multi-bug version of the grep program. A 5-bug version of the grep program is created by combining the bugs in five single fault versions. As a result, we no longer have five different versions of the grep program, but rather operate on a single version that contains five bugs. Failed tests are manually grouped into five fault-focused clusters. A more robust clustering technique will be developed in our future study; however, that is beyond the scope of this paper. Similar to the single fault versions, we discount non-executable statements and combine multi-line statements leaving us with a version with 3306 executable statements.

Without loss of generality, we apply Heuristic III(c) to illustrate the application of the method. Data are presented in a fashion similar to the study on single fault programs and following the same conventions. Fig. 10 gives the comparison between Heuristic III(c) and Tarantula for the multi-bug version of grep. It is clear that not only is Heuristic III(c) more effective than Tarantula for both best and worst cases but also that Heuristic III(c) Worst is more effective than TBest.

Tables 15 and 16 reveal that Heuristic III(c) yields better results than Tarantula by having a higher effectiveness compared to Tarantula both in terms of the pair-wise comparison and the total number of statements that are required to be examined.

5. Discussions

In this section, we expand on some points relevant to our fault localization method.

Table 15
Pair-wise comparison between Heuristic III(c) and Tarantula.

	Heuristic III(c) Best versus TBest	Heuristic III(c) Worst versus TWorst
More effective	3	3
Same effectiveness	2	2
Less effective	0	0

5.1. Impact of α and $\chi_{F/S}$ in Eq. (4) on the effectiveness of fault localization

In practice, during testing we have many more successful tests than failed tests. For example, a statement $\mathscr{S}$ containing a bug when executed may only cause a small number of failures (say, 10 out of 1000 executions). We assume that no test case is used more than once. Then, we have 10 failed tests and 990 successful tests for $\mathscr{S}$ ($\mathscr{N}_F = 10$ and $\mathscr{N}_S = 990$). Suppose also the total number of failed and successful tests for this bug is 10 and 990, respectively ($\Phi_F = 10$ and $\Phi_S = 990$).

If each failed and successful test make the same contribution towards the debugging (as suggested by Heuristic II), the suspiciousness of $\mathscr{S}$ has a value of $10 - 990 = -980$. This implies S has a very low priority to be examined for locating the fault. Results of our case studies reported in Section 3.4 have shown that this is not a good heuristic. The main reason is that the contribution from the successful tests is overweighted. To solve this problem, Heuristic III as described by Eq. (3) is proposed.

To facilitate the discussion, we chose the same parameter values as Eq. (4). That is, the failed and the successful tests are divided into three groups each. Using the above example and following the same way as described in Section 2 on how tests are divided, we have 2, 4 and 4 tests in the first, second, and third failed groups, respectively ($n_{F,1} = 2$, $n_{F,2} = 4$, and $n_{F,3} = 4$). The total contribution from all the failed tests that execute S is $\sum_{i=1}^{\mathscr{N}_F} c_{F,i} = (1 \times 2 + 0.1 \times 4 + 0.01 \times 4) = 2.44$. Similarly, we have 1, 4, and 985 tests in the first, second, and third successful groups, respectively ($n_{S,1} = 1$, $n_{S,2} = 4$, and $n_{S,3} = 985$). The total contribution from all the successful tests that execute $\mathscr{S}$ is $\sum_{i=1}^{\mathscr{N}_S} c_{S,i} = (1 \times 1 + 0.1 \times 4) +$"contribution from the third successful group" = 1.4 + "contribution from the third successful group". We observe that Φ_S is 99× of Φ_F. Under this condition, one way to ensure $\sum_{i=1}^{\mathscr{N}_S} c_{S,i} < \sum_{k=1}^{\mathscr{N}_F} c_{F,k}$ is to make the contribution of each test in the third successful group less than $\chi_{F/S} = \frac{\Phi_F}{\Phi_S}$ of the contribution of each test in the third failed test. In our case, the former is less than $\frac{10}{990} = \frac{1}{99}$ of the latter. Since the contribution of each test in the third failed group is 0.01, the contribution of each test in the third successful group is $0.01 \times \frac{1}{99}$. As a result, $\sum_{i=1}^{\mathscr{N}_S} c_{S,i} = (1 \times 1 + 0.1 \times 4 + 0.01 \times \frac{1}{99} \times 995) = 1.50$, and the suspiciousness of $\mathscr{S}$ is $2.44 - 1.50 = 0.94$ in contrast to -980 computed by using Heuristic II. We observe that the requirement $\sum_{i=1}^{\mathscr{N}_S} c_{S,i} < \sum_{k=1}^{\mathscr{N}_F} c_{F,k}$ is satisfied. In addition, $\mathscr{S}$ is ranked as a much more suspicious statement by Heuristic III(a) than Heuristic II. This is consistent with our intuition.

We can further reduce $\sum_{i=1}^{\mathscr{N}_S} c_{S,i}$ by using a smaller α (the scaling factor in Eq. (4)). In Heuristics III(b) and (c), α is set to 0.001 and 0.0001, respectively. The suspiciousness of $\mathscr{S}$ is $2.44 - (1 \times 1 + 0.1 \times 4 + 0.001 \times \frac{1}{99} \times 995) = 1.030$ and $2.44 - (1 \times 1 + 0.1 \times 4 + 0.0001 \times \frac{1}{99} \times 995) = 1.039$, respectively. In both cases, we have $\sum_{i=1}^{\mathscr{N}_S} c_{S,i} < \sum_{k=1}^{\mathscr{N}_F} c_{F,k}$.

An interesting observation is that the smaller the α, the more suspicious the statement S. As for which α is the optimal, it depends on several factors such as the structure of the program, the nature of the bug, and the test set used for testing the program.

Finally, we would like to emphasize that there are two fundamental principles for Heuristic III: (1) $c_{F,1} \geq c_{F,2} \geq c_{F,3} \geq \ldots \geq c_{F,\mathscr{N}_F}$ & $c_{S,1} \geq c_{S,2} \geq c_{S,3} \geq \ldots \geq c_{S,\mathscr{N}_S}$, and (2) $\sum_{i=1}^{\mathscr{N}_S} c_{S,i} < \sum_{k=1}^{\mathscr{N}_F} c_{F,k}$. When we describe Heuristic III, we should refer to Eq. (3) rather than Eq. (4) as the latter is just an instance of the former. However, to apply this Heuristic in practice, we need to assign values to certain parameters. Different values can be used for different scenarios as long as the above two principles are not violated. The values selected for our case studies have been proven to be good since the best (and worst) effectiveness computed by Heuristics III(a), (b), and (c) with these parameters is better than the respective effectiveness of Tarantula. Also, our experimental data show that smaller values of α give a better effectiveness. However, the improvement is reduced as we decrease α.

5.2. Additional considerations

5.2.1. Bugs due to missing code

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

5.2.2. Bugs due to a sequence of test cases

Sometimes a program fails not because of the current test but because of a previous test which fails to set up an appropriate execution environment for the current test, then we should bundle these two test cases together as one single failed test. For example, consider two test cases t_α and t_β which are executed successively in this order. Assume also a bug in the program causes t_α to incorrectly modify values at certain memory locations but does not result in a failure of t_α. These incorrect values are to be referenced by t_β and are the only source of the failure for t_β. Under this situation we should combine t_α and t_β as a single failed test.

5.2.3. Limitation and threats to validity

There are several threats to the validity of the experiments and their derived results which are presented in this paper. In this section we discuss and address some of them.

We would also like to emphasize that like any other fault localization methods, the effectiveness of our methods varies for different programs, bugs, and test cases. While it is true that the

Table 16
Total number of statements examined by Heuristic III(c) and Tarantula.

	Heuristic III(c) Best	Heuristic III(c) Worst	TBest	TWorst	Heuristic III(c) Best/TBest (%)	Heuristic III(c) Worst/TWorst (%)
Number of statements	734	1192	1349	1807	54.41	65.97

Code examined	Percentage of faults detected				
	Heuristic III(a)	Heuristic III(b)	Heuristic III(c)	SOBER	Liblit05
10%	59.69%	68.22%	68.22%	52.31%	40%
20%	70.54%	79.07%	79.07%	73.85%	63%
30%	75.97%	88.37%	89.92%	77%	70.5%
40%	85.27%	93.80%	94.57%	81%	73%
50%	94.57%	96.12%	96.12%	82%	76%
60%	94.57%	98.45%	98.45%	83%	77.5%
70%	96.12%	99.22%	99.22%	85%	88%
80%	97.67%	100%	100%	88%	90%
90%	100%			90%	91%
100%				100%	100%

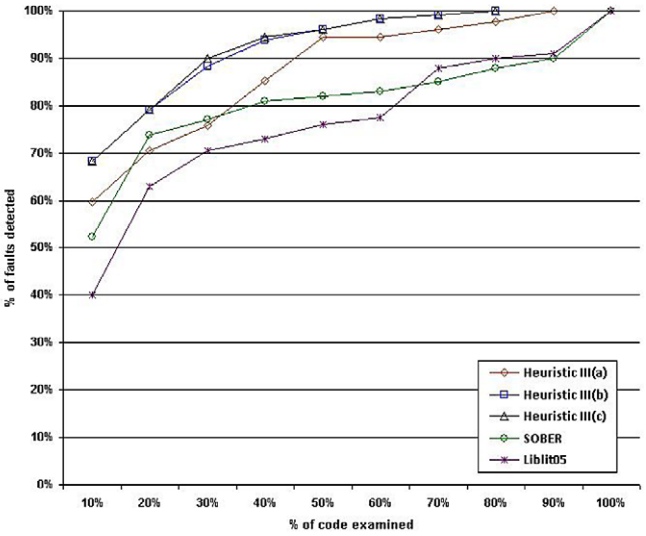


Fig. 11. Comparison of Heuristic III against SOBER and Liblit05 on the Siemens suite.

evaluation of the methods presented here is based on empirical data and therefore we may not be able to generalize our results to all programs; it is for this reason that we observed the effectiveness of the methods across such a broad spectrum of programs. Each of the subject programs varies greatly from the other with respect to size, function, number of faulty versions studied, etc. This allows us to have greater confidence in the applicability of our fault localization methods to different programs and the superiority of the results. The coverage measurement tools (such as whether the runtime trace can be correctly collected even if a program execution is crashed due to a segmentation fault) and environments (including compilers, operating systems, hardware platforms, etc.) are also expected to have an impact.

This paper also assumes perfect bug detection which may not always hold in practice. Our fault localization methods provide a programmer with a ranked list of statements sorted in order of their suspiciousness which may then be examined in order to identify the faults in the statements. We assume that if a programmer examines a faulty statement, the programmer shall consequently also identify the corresponding fault. Additionally, the programmer shall not identify a non-faulty statement incorrectly as faulty. Should such perfect bug detection not hold, then the amount of code that needs to be examined to detect the bug may increase. However, such a concern applies equally to other fault localization methods such as those discussed in Section 6.

6. Related studies

Many studies have reported different fault localization methods. In this section we focus on Tarantula (Jones and Harrold, 2005) and a few other related methods. In addition, we also compare the effectiveness of our method with that of SOBER (Liu et al., 2006) and Liblit05 (Liblit et al., 2005), two statistical debugging methods published in the recent literature.

6.1. Tarantula and a few related fault localization methods

Similar to our method, Tarantula also uses the coverage and execution results to compute the suspiciousness of each statement as $X/(X+Y)$ where X = (number of failed tests that execute the statement)/(total number of failed tests) and Y = (number of successful tests that execute the statement)/(total number of successful tests). A recent study (Jones et al., 2007) applies Tarantula to programs with multiple bugs. Refer to Section 4 for more details.

Renieris and Reiss (2003) propose a nearest-neighbor debugging approach that contrasts a failed test with another successful test which is most similar to the failed one in terms of the "distance" between them. In this approach, the execution of a test is represented as sequence of basic blocks that are sorted by their execution times. If a bug is in the difference set, it is located. For a bug that is not contained in the difference set, the approach can continue the bug localization by first constructing a program dependence graph and then including and checking adjacent unchecked nodes in the graph step by step until all the nodes in the graph are examined.

Two methods based on the program spectra in Renieris and Reiss (2003) are the set-union method and the set intersection method. The set-union computes the set difference between the program spectra of a failed test and the union spectra of a set of successful tests. It focuses on the source code that is executed by the failed test but not by any of the successful tests. Such code is more suspicious than others. The set intersection method excludes the code that is executed by all the successful tests but not by the failed test.

Cleve and Zeller (2005) report a program state-based debugging approach, cause transition, to identify the locations and times where a cause of failure changes from one variable to another. This approach is based on their previous research on delta debugging (Zeller, 2002; Zeller and Hildebrandt, 2002). An algorithm named *cts* is proposed to quickly locate cause-transitions in a program execution. A potential problem of the cause transition approach is that the cost of such an approach is relatively high; there may exist thousands of states in a program execution, and delta debugging at each matching point requires additional test runs to narrow down the causes.

Abreu et al. (2007) investigate the diagnostic accuracy of spectrum-based fault localization as a function of several parameters such as the quality and quantity of the program spectra collected during test case execution. Their study of the Jaccard, the Tarantula and the Ochiai similarity coefficients on the Siemens suite suggests that the superior performance of a particular coefficient is largely independent of test case design. Wang et al. (2009) propose an approach to address the problem of co-incidental correctness (when

a fault is executed but a corresponding failure is not detected) in the context of fault localization. The authors make use of control flow and data flow patterns, which may appear when the execution of faults triggers failures, to refine code coverage which in turn can strengthen the correlation between program failure and faulty statement coverage. Thus, the study of Wang et al. (2009) seeks to refine the inputs to coverage-based fault localization (CBFL) techniques leading to more effective fault localization.

6.2. SOBER and Liblit05

We compare our method with two statistical debugging methods reported in recent publications. The first one is by Liblit et al. (2005) who presented a statistical debugging algorithm for deployed programs (referred to hereafter as Liblit05). The second is by Liu et al. (2006) who proposed the SOBER model to rank suspicious predicates. The major difference between our method, SOBER, and Liblit05 is that we rank suspicious statements, whereas the last two rank suspicious predicates for fault localization. For them, the corresponding statements of the top k predicates are taken as the initial set to be examined for locating the fault. As suggested by Jones and Harrold (2005), Liblit05 provides no way to quantify the ranking for all statements. An ordering of the predicates is defined, but the approach does not expand on how to order statements related to any bug that lies outside a predicate. For SOBER, if the bug is not in the initial set of statements, additional statements have to be included by performing a breadth-first search on the corresponding program dependence graph. However, this search is not required using our method as all the statements of the program are ranked based on their suspiciousness (Section 2). This suspiciousness ranking for each statement can be computed very efficiently once the number of successful and failed tests that execute each statement has been determined.

Below, we present a quantitative comparison on the effectiveness using the Siemens suite between our method, SOBER and Liblit05. From Section 3.4 and Fig. 11, we observe that in the best case Heuristic III(a) can identify 59.69%, and Heuristics III(b) and (c) can each identify 68.22% of the bugs, respectively, by examining no more than 10% of the code. According to the data reported in Liu et al. (2006), of the 130 bugs in the Siemens suite, SOBER can help programmers locate 52.31% and Liblit05 40% of the bugs, by examining at most 10% of the code in the best case.

Furthermore, we are able to retrieve data from the curves in Fig. 4b of Liu et al. (2006) about the best case effectiveness of SOBER and Liblit05.[7] Fig. 11 gives a comparison between Heuristics III(a), (b), (c), SOBER and Liblit05 on the Siemens suite.

We make the following observations about the best case effectiveness

- Heuristics III(a), (b) and (c) are all better than Liblit05.
- Heuristic III(a) is worse than SOBER when approximately 15% to 30% of the code is examined, but is better than or equal to SOBER for all other percentages.
- Heuristics III(b) and (c) are both better than SOBER.
- Heuristics III(b) and (c) are better than Heuristic III(a).
- Heuristic III(c) is better than or equal to Heuristic III(b).

It is clear that with respect to the Siemens suite, Heuristics III(a), (b) and (c) are more effective than SOBER and Liblit05 in locating faults for almost all the cases.

[7] Since the exact numbers are not reported in Liu et al. (2006), we had to get them directly from the curves. To minimize any possible human error, two PhD students retrieved the data independently. These data are then cross validated for their accuracy against the curves.

7. Conclusions and future work

We propose a family of code coverage-based heuristics for fault localization to help programmers effectively locate faults. Heuristic I is straightforward and intuitive, but by itself is limited because it only takes into account the contribution of failed tests. Heuristic II is proposed to overcome this drawback by including the contribution of successful tests as well. However, it may assign undue importance to this contribution. In light of this observation, Heuristic III is proposed with two principles. First, while computing the suspiciousness of a statement, the total contribution of the successful tests should be less than that of the failed (namely, $\sum_{i=1}^{\mathcal{N}_S} c_{S,i} < \sum_{k=1}^{\mathcal{N}_F} c_{F,k}$). Second, the contribution of the first successful test case is larger than or equal to that of the second successful test case, which in turn is larger than or equal to that of the third successful test case and so on. The same is also applied for failed test cases (namely, $c_{F,1} \geq c_{F,2} \geq c_{F,3} \geq \ldots \geq c_{F,\mathcal{N}_F}$ & $c_{S,1} \geq c_{S,2} \geq c_{S,3} \geq \ldots \geq c_{S,\mathcal{N}_S}$) Different values can be assigned to parameters in Eq. (3). Three α's (0.01, 0.001 and 0.0001) are used in Heuristics III(a), (b) and (c), respectively. A discussion on the impact of α and $\chi_{F/S}$ on the effectiveness of fault localization is presented in Section 5.1.

Results from our studies on six sets of programs suggest that Heuristics I and II are not good for fault localization, but Heuristics III(a), (b) and (c) are effective. They can perform better than other debugging methods (including set-union, set intersection, nearest-neighbor, cause-transitions, Tarantula, Liblit05 and SOBER) in locating program faults, as they lead to the examination of a smaller percentage of the code than other methods before the faults are located. A discussion of extending our method to programs with multiple bugs is also reported in Section 4.

For future work, we would like to explore the benefits of applying the proposed fault localization heuristics in practice to large-scale industry software. We are currently working with engineers from companies such as Raytheon and Lockheed Martin Aeronautics Company to study how real-world developers can benefit from our method. We will also develop more robust clustering techniques for grouping failed tests with respect to each fault. In addition, more studies examining the impact of α and $\chi_{F/S}$ on our method are currently underway.

Acknowledgement

We wish to thank Yu Qi and Lei Zhao for proving valuable insights while this paper was a work in progress. We are also extremely grateful to the anonymous referees whose comments and suggestions were critical to the improvement of the quality of the paper.

References

Abreu, R., Zoeteweij, P., van Gemund, A.J.C., 2007. On the accuracy of spectrum-based fault localization. In: Proceedings of the Testing: Academic and Industrial Conference – Practice and Research Techniques, September 2007, pp. 89–98.

Agrawal, H., DeMillo, R.A., Spafford, E.H., 1996. Debugging with dynamic slicing and backtracking. Software – Practice and Experience 23 (6), 589–616 (June).

Agrawal, H., Horgan, J.R., 1990. Dynamic program slicing. In: Proceedings of the ACM SIGPLAN'90 Conference on Programming Language Design and Implementation, June 1990.

Agrawal, H., Horgan, J.R., Wong, W.E., et al., 1998. Mining system tests to aid software maintenance. IEEE Computer 31 (7), 64–73 (July).

Agrawal, H., Horgan, J.R., London, S., Wong, W.E., 1995. Fault localization using execution slices and dataflow tests. In: Proceedings of the 6th IEEE International Symposium on Software Reliability Engineering, Toulouse, France, October 1995, pp. 143–151.

Andrews, J.H., Briand, L.C., Labiche, Y., 2005. Is mutation an appropriate tool for testing experiments? In: Proceedings of the 27th International Conference on Software Engineering, St. Louis, Missouri, USA, May 2005, pp. 402–411.

Cleve, H., Zeller, A., 2005. Locating causes of program failures. In: Proceedings of the 27th International Conference on Software Engineering (ICSE 2005), St. Louis, Missouri, USA, May 2005, pp. 342–351.

Ellsberger, J., Hogrefe, D., Sarma, A., 1997. SDL: Formal object-oriented language for communicating systems. Prentice Hall.
Horgan, J.R., London, S.A., 1991. Data flow coverage and the C Language. In: Proceedings of the 4th Symposium on Software Testing, Analysis, and Verification, Victoria, British Columbia, Canada, October 1991, pp. 87–97.
Hutchins, M., Foster, H., Goradia, T., Ostrand, T., 1994. Experiments on the effectiveness of dataflow- and controlflow-based test adequacy criteria. In: Proceedings of the 16th International Conference on Software Engineering, Sorrento, Italy, May 1994, pp. 191–200.
Jones, J.A., Harrold, M.J., 2005. Empirical evaluation of the Tarantula automatic fault-localization technique. In: Proceedings of the 20th IEEE/ACM International Conference on Automated Software Engineering (ASE 2005), Long Beach, CA, USA, November 2005, pp. 273–282.
Jones, A., Bowring, J., Harrold, M.J., 2007. Debugging in parallel. In: Proceedings of the 2007 International Symposium on Software Testing and Analysis, London, UK, July 2007, pp. 16–26.
Korel, B., Laski, J., 1988. Dynamic program slicing. Information Processing Letters 29 (3), 155–163 (October).
Liblit, B., Naik, M., Zheng, A.X., Aiken, A., Jordan, M.I., 2005. Scalable statistical bug isolation. In: Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation, Chicago, IL, June 2005, pp. 15–26.
Lyle, J.R., Weiser, M., 1987. Automatic program bug location by program slicing. In: Proceedings of the 2nd International Conference on Computer and Applications, Beijing, China, June 1987, pp. 877–883.
Liu, C., Fei, L., Yan, X., Han, J., Midkiff, S.P., 2006. Statistical debugging: a hypothesis testing-based approach. IEEE Transactions on Software Engineering 32 (10), 831–848 (October).
Liu, C., Han, J., 2006. Failure proximity: a fault localization-based approach. In: Proceedings of the 14th ACM SIGSOFT International Symposium on Foundations of Software Engineering, Portland, Oregon, USA, November 2006, pp. 46–56.
Renieris, M., Reiss, S.P., 2003. Fault localization with nearest neighbor queries. In: Proceedings of the 18th IEEE International Conference on Automated Software Engineering, Montreal, Canada, October 2003, pp. 30–39.
Podgurski, A., Leon, D., Francis, P., Masri, W., Minch, M., Sun, J., Wang, B., 2003. Automated support for classifying software failure reports. In: Proceedings of the 25th International Conference on Software Engineering, Portland, Oregon, USA, May 2003, pp. 465–475.
The Siemens Suite, January 2007. <http://www-static.cc.gatech.edu/aristotle/Tools/subjects/>.
Vessey, I., 1985. Expertise in debugging computer programs. International Journal of Man-Machine Studies: A Process Analysis 23 (5), 459–494.
Wang, X., Cheung, S.C., Chan, W.K., Zhang, Z., 2009. Taming coincidental correctness: refine code coverage with context pattern to improve fault localization. In: Proceedings of the 31st IEEE International Conference on Software Engineering, Vancouver, Canada, May 2009, pp. 45–55.
Weiser, M., 1982. Programmers use slices when debugging. Communications of the ACM 25 (7), 446–452 (July).
Wong, W.E., Li, J.J., 2005. An integrated solution for testing and analyzing Java applications in an industrial setting. In: Proceedings of the 12th IEEE Asia-Pacific Software Engineering Conference (APSEC), Taipei, Taiwan, December 2005, pp. 576–583.
Wong, W.E., Horgan, J.R., London, S., Mathur, A.P., 1998. Effect of test set minimization on fault detection effectiveness. Software-Practice and Experience 28 (4), 347–369 (April).
Wong, W.E., Qi, Y., 2006. Effective program debugging based on execution slices and inter-block data dependency. Journal of Systems and Software 79 (7), 891–903 (July).
Wong, W.E., Qi, Y., 2009. BP neural network-based effective fault localization. International Journal of Software Engineering and Knowledge Engineering 19 (4), 573–597 (June).
Wong, W.E., Sugeta, T., Qi, Y., Maldonado, J.C., 2005. Smart debugging software architectural design in SDL. Journal of Systems and Software 76 (1), 15–28 (April).
Wong, W.E., Shi, Y., Qi, Y., Golden, R., 2008a. Using an RBF neural network to locate program bugs. In: Proceedings of the 19th IEEE International Symposium on Software Reliability Engineering (ISSRE), Seattle, USA, November 2008, pp. 27–38.
Wong, W.E., Wei, T., Qi, Y., Zhao, L., 2008b. A Crosstab-based statistical method for effective fault localization. In: Proceedings of the First International Conference on Software Testing, Verification and Validation (ICST), Lillehammer, Norway, April 2008, pp. 42–51.
χSuds User's manual, 1998. Telcordia Technologies.
Zeller, A., 2002. Isolating cause-effect chains from computer programs. In: Proceedings of the 10th ACM SIGSOFT Symposium on Foundations of Software Engineering, Charleston, South Carolina, USA, November 2002, pp. 1–10.
Zeller, A., Hildebrandt, R., 2002. Simplifying and isolating failure-inducing input. IEEE Transactions on Software Engineering 28 (2), 183–200 (February).
Zhang, X., Gupta, N., Gupta, R., 2006. Locating faults through automated predicate switching. In: Proceeding of the 28th International Conference on Software Engineering, Shanghai, China, May 2006, pp. 272–281.
Zheng, A.X., Jordan, M.I., Liblit, B., Naik, M., Aiken, A., 2006. Statistical debugging: simultaneous identification of multiple bugs. In: Proceedings of the 23rd International Conference on Machine Learning, Pittsburgh, Pennsylvania, June 2006, pp. 1105–1112.

W. Eric Wong received his M.S. and Ph.D. in Computer Science from Purdue University, West Lafayette, Indiana. He is currently an associate professor in the Department of Computer Science at the University of Texas at Dallas. Dr. Wong is a recipient of the Quality Assurance Special Achievement Award from Johnson Space Center, NASA (1997). Prior to joining UT-Dallas, he was with Telcordia Technologies (formerly Bell Communications Research a.k.a. Bellcore) as a Senior Research Scientist and as the project manager in charge of the initiative for Dependable Telecom Software Development. Dr. Wong's research focus is on the technology to help practitioners produce high quality software at low cost. In particular, he is doing research in the areas of software testing, debugging, safety, reliability, and metrics. He has received funding from such organizations as NSF, NASA, Avaya Research, Texas Instruments, and EDS/HP among others. He has published over 120 refereed papers in journals and conference/workshop proceedings. Dr. Wong has served as special issue guest editor for six journals and as general or program chair for many international conferences. He also serves as the Secretary of ACM SIGAPP and is on the Administrative Committee of the IEEE Reliability Society.

Vidroha Debroy received his B.S. in Software Engineering and his M.S. in Computer Science from the University of Texas at Dallas. He is currently a PhD student in Computer Science at UT-Dallas. His interests include software testing and fault localization, program debugging and automated and semi-automated ways to repair software faults. He is a student member of the IEEE.

Byoungju Choi is a professor in the Department of Computer Science and Engineering at Ewha Womans University, Seoul, Korea. She received her B.S. in Mathematics from Ewha Womans University and her M.S. and Ph.D. in Computer Science from Purdue University, USA. Her research interests are in software engineering with particular emphasis on software testing, embedded software testing, software and data quality, and software process improvement.