Research Article

Program Spectra Analysis with Theory of Evidence

Rattikorn Hewett

Department of Computer Science, Texas Tech University, Lubbock, TX 79409-3104, USA

Correspondence should be addressed to Rattikorn Hewett, rattikorn.hewett@ttu.edu

Received 30 August 2011; Revised 4 January 2012; Accepted 8 January 2012

Academic Editor: Chin-Yu Huang

Copyright © 2012 Rattikorn Hewett. This is an open access article distributed under the Creative Commons Attribution License, which permits unrestricted use, distribution, and reproduction in any medium, provided the original work is properly cited.

This paper presents an approach to automatically analyzing *program spectra*, an execution profile of program testing results for fault localization. Using a mathematical theory of evidence for uncertainty reasoning, the proposed approach estimates the likelihood of faulty locations based on evidence from program spectra. Our approach is theoretically grounded and can be computed online. Therefore, we can predict fault locations immediately after each test execution is completed. We evaluate the approach by comparing its performance with the top three performing fault localizers using a benchmark set of real-world programs. The results show that our approach is at least as effective as others with an average *effectiveness* (the reduction of the amount of code examined to locate a fault) of 85.6% over 119 versions of the programs. We also study the quantity and quality impacts of program spectra on our approach where the quality refers to the spectra *support* in identifying that a certain unit is faulty. The results show that the effectiveness of our approach slightly improves with a larger number of failed runs but not with a larger number of passed runs. Program spectra with support quality increases from 1% to 100% improves the approach's effectiveness by 3.29%.

1. Introduction

Identifying location of faulty software is notoriously known to be among the most costly and time-consuming process in software development [1, 2]. As software gets larger and more complex, the task can be daunting even with the help of debugging tools. Over the decades, many approaches to software fault localization have been studied including diagnostic reasoning [3], program slicing [4], nearest neighbor [5], and statistical analysis [6, 7].

Recent fault localization techniques have focused on automatically analyzing program behaviors observed from the execution of a suite of test cases on the tested program called *program spectra* [8]. For each run of the test case, certain program units (e.g., statements or blocks of code) are executed and result in either a *passed test* (*run*, or *execution*) when the output of the program's execution is the same as the expected output, or a *failed test*, otherwise. A collection of program spectra contains execution profiles that indicate which part of the program is involved in each test run and whether it is passed or failed.

Spectrum-based fault localization basically tries to identify the part of the program whose activity correlates most

with the resulting passed or failed test runs. Most existing spectrum-based approaches rely on similarity measures to locate faulty software units by identifying the units that most resemble the spectra error outcomes [9–15]. The technique has been used for fault localization in various applications including the Pinpoint tool [12] for large dynamic online transaction processing systems and AMPLE [16] for objected-oriented software. Spectrum-based fault localization is relatively efficient to compute and does not require modeling of the program under investigation. Therefore, it is a popular fault localization technique that can easily be integrated into testing procedures [10].

The current top three performing spectrum-based fault localizers include *Ochiai* [14], *Jaccard* [12], and *Tarantula* [13]. *Tarantula* uses a heuristic function adapted from a visualization technique while *Jaccard* and *Ochiai* employ different similarity measures, both of which are widely used in other domains such as biology and ecology. While these approaches are useful, most lack theoretical foundation and the ability to immediately incorporate new testing results into the fault localization process. Furthermore, they are not easily extensible to new findings of contributing factors. Our research aims to alleviate these shortcomings.

This paper presents a spectrum-based fault localization technique for pinpointing locations of faulty software units. The approach employs the theory of evidence called Dempster-Shafer Theory [17] for uncertainty reasoning to estimate the likelihood of faulty locations based on evidence gathered from program spectra. Our approach is theoretically grounded and computed online instead of batch. Thus, it allows the prediction of fault locations to be identified immediately as the execution of each test case is completed without having to wait to collect a set of program spectra that is large enough to be statistically valid. Our contribution also includes a study of the influences of the theory of evidence on fault localization effectiveness as well as the influences of the quantity and quality of the program spectra used in the analysis.

The rest of the paper is organized as follows: Section 2 describes preliminary concepts and terminology including basic mechanisms for software fault localization, the three spectrum-based fault localizers used in our comparison study, and the Dempster-Shafer Theory along with its fundamental elements. Section 3 presents our approach to program spectra analysis, an illustration on a small example program, and some characteristic comparisons. Section 4 evaluates the proposed technique and discusses the empirical study using a set of standard benchmarks to compare the proposed method against the other three prominent software fault localizers. Section 5 presents an empirical study to answer questions whether the performance of the proposed approach relies on the quality or quantity of the program spectra or not. Section 6 discusses the work related to fault localization and the method proposed in this paper. Section 7 concludes the paper.

2. Preliminaries

This section describes terms, concepts of spectrum-based fault localization technique, the three fault localizers and the basic foundations of the mathematical theory of evidence.

2.1. Concepts, Terms, and Notations. Following the terminology in [10], a *software failure* occurs when the actual program output, for a given input, deviates from the corresponding specified output. *Software errors*, however, define defects that may or may not cause a failure. Thus, in practice, we may not be able to detect all errors. Defects that result in failures are referred to as *software faults* (or *bugs*).

Program spectra [8] are execution profiles of the program resulting from test runs. In particular, suppose we run m test cases on a program of n units (e.g., statements, blocks, and modules). The *hit spectra* can be represented as an $m \times n$ matrix $R = (r_{ij})$, where $r_{ij} = 1$ if test i involves execution of unit j of the program, otherwise it is 0 (i.e., if test i does not involve execution of unit j). In addition, for each run of test i , we define a corresponding *error* e_i to be 1 if the test failed, and 0, otherwise (i.e., when the test was passed or successful). Program spectra include both the hit spectra and the error. To collect program spectra, we run a suite of test cases and observe execution results where the test can be performed at

TABLE 1: Different types of test run frequency count.

Software Unit	Test run	
	Passed	Failed
Not Executed	a_{00}	a_{01}
Executed	a_{10}	a_{11}

various levels of software units (e.g., code lines or statements, blocks of code, or modules). Much research has studied fault localization in code blocks [9, 15]. Though we illustrate our approach at a code statement (or line) level, the approach is general for application in any level of software unit.

Conceptually, *fault localization* is an attempt to identify a software unit whose behaviors in all relevant test cases are most similar to the errors observed in a given program spectra. To do this, the following notations are defined. For a given test run i and a software unit j , let p and q be a binary variable signifying whether run i involves execution of unit j , and, respectively, whether run i fails or not. Here p is 1 if run i involves execution of unit j , and 0, otherwise. On the other hand, q is 1 if run i fails and 0, otherwise. Next we relate p and q to observations in the program spectra.

For each software unit j , we define the frequency of runs $a_{pq}(j) = |\{i \mid r_{ij} = p, e_i = q\}|$. Recall that r_{ij} represents the result of test run i on unit j (whether it executes unit j or not), and e_i represents the output error of test run i (whether the test is passed or failed). Thus, for a given software unit, we can summarize the interpretations of all possible cases of a_{pq} 's in Table 1. For example, $a_{01}(j)$ represents the number of *failed* test runs (i.e., $q = 1$) that *did not execute* (i.e., $p = 0$) unit j . Note that in general $a_{00}(j)$ is not of interest to fault localization since a successful test run that does not involve the execution of the software unit does not provide any useful information for detecting faults. It does not raise any suspicion that the unit under investigation is faulty nor does it confirm that it is not faulty. On the other hand, $a_{11}(j)$ is an important indicator for identifying faulty units.

2.2. Spectrum-Based Fault Localizers. In spectrum-based fault localization techniques, software units are ranked based on their corresponding *similarity coefficients* (e.g., see [5, 10]). A similarity coefficient of a software unit measures how closely the execution runs of the test cases that involve the considering unit resemble the errors observed [9]. The software unit that has a high similarity to the output errors is assumed to have a high probability that the unit would be the cause of such errors. Thus, a unit with a larger similarity-coefficient value would rank higher in terms of its chance to be faulty. Let S_i denote a similarity coefficient for software unit i .

Next we describe the current top three fault localizers that mainly differ by the similarity coefficient S_i . A popular Tarantula [13] has been shown to be the best performing spectrum-based fault localizer in [18]. Abreu et al. [9] have recently shown that the *Jaccard* coefficient [12] and *Ochiai* coefficient [14] marginally outperform Tarantula. Most similarity coefficients utilize the frequency of occurrences of

the result of the test execution that involves a certain code statement. We now describe them in more detail.

Tarantula. The similarity coefficient in Tarantula is adapted from a formula for displaying the color of each program statement to visualize fault locations. For a given code statement i , $\%passed(i)$ ($\%failed(i)$) represents, in percentage, a ratio of the number of passed (failed) test cases that executed i to the total number of the passed (failed) test cases in the overall test suite. Tarantula quantifies *color* value of a given code statement by the following formula:

$$\text{color} = L + \left(\frac{\%passed}{\%passed + \%failed} \right) * (H - L), \quad (1)$$

where L and H represent low end and a high end color values of the color spectrum, respectively. For example, if a program statement is executed by 10% of the passed test cases and 20% of the failed test cases, its color will be 1/3 of the way from pure red (low end color of value zero) to pure green (high end color of value 120), thus in between red and yellow making it an orange color (of value 40).

Adapting from the above color scheme, the similarity coefficient is defined as follows:

$$S_i = \frac{\%failed(i)}{\%failed(i) + \%passed(i)}. \quad (2)$$

By using the notation introduced in Section 2.1, we obtain the following:

$$\begin{aligned} \%failed(i) &= \frac{a_{11}(i)}{a_{11}(i) + a_{01}(i)}, \\ \%passed(i) &= \frac{a_{10}(i)}{a_{10}(i) + a_{00}(i)}. \end{aligned} \quad (3)$$

The numerator in S_i is opposite from that of the color scheme because Tarantula ranks the suspicion of faulty units from those with the highest to the lowest values of S_i 's. Thus, the higher value of S_i indicates that statement i is more likely to be faulty. Faulty likelihood is influenced by $\%failed$. On the contrary, in the color scheme, the lowest color value is represented in pure red to signify that the unit under investigation is the most likely to be faulty.

Jaccard. The Jaccard similarity coefficient [19] is a simple metric that has been used for comparing two binary data objects whose variable values are not equally important (e.g., a positive disease test result is more crucial than a negative one). Chen et al. [12] have applied the Jaccard similarity coefficient for fault localization in a pinpoint tool. The Jaccard coefficient is defined as

$$S_i = \frac{a_{11}(i)}{a_{11}(i) + a_{01}(i) + a_{10}(i)}. \quad (4)$$

As shown in the formula, the numerator of the Jaccard similarity coefficient uses $a_{11}(i)$ to compare the similarity in frequency of the test cases of interest (i.e., failed tests that execute line i). Furthermore, the denominator omits

$a_{00}(i)$, which provides no valuable information with respect to locating faulty lines. Although the Jaccard coefficient has been shown to improve fault localization performance over Tarantula [9], the difference is marginal. Additional experiments are required to evaluate and draw conclusions.

Ochiai. The Ochiai coefficient [20] has been applied to various domains including ecology and molecular biology [14]. The coefficient is defined as the following:

$$S_i = \frac{a_{11}(i)}{\sqrt{(a_{11}(i) + a_{01}(i)) * (a_{11}(i) + a_{10}(i))}}. \quad (5)$$

The Ochiai coefficient uses the same contributing factors to measure similarity as those of Jaccard's. However, the denominator has more complex computation. In fault localization application, Abreu et al.'s empirical study [9] has shown that Ochia coefficient yields more superior results than those obtained from Jaccard coefficient. However, there is no intuitive explanation given.

2.3. Mathematical Theory of Evidence. Work in spectrum-based fault localization has mostly concentrated on specifying appropriate similarity coefficients using the information extracted from program spectra. This is quite different from our approach. To provide a theoretical background of the proposed research, we describe the mathematical theory of evidence, also known as the *Dempster-Shafer (D-S) Theory* [17]. The D-S theory allows probability assignment to a set of atomic elements rather than an atomic element. Thus, the D-S theory can be viewed as a generalization of Bayesian probability theory that can explicitly represent ignorance as well as uncertainty [21].

Let $\mathcal{U}$ be a finite set of all hypotheses (atomic elements) in a problem domain. A *mass function* m provides a probability assignment to any $A \subseteq \mathcal{U}$, where $m(\emptyset) = 0$ and $\sum_{A \subseteq \mathcal{U}} m(A) = 1$. The mass $m(A)$ represents a belief *exactly* on A . For example, $\mathcal{U} = \{\text{faulty}, \sim \text{faulty}\}$ represents a set of two hypotheses of a suspect being faulty and nonfaulty, respectively. In such a case, the property of the mass function implies that $m(\{\text{faulty}\}) + m(\{\sim \text{faulty}\}) + m(\{\text{faulty}, \sim \text{faulty}\}) = 1$, as $m(\emptyset) = 0$. Thus, mass function is not the same as probability. When there is no information regarding $\mathcal{U}$, $m(\{\text{faulty}, \sim \text{faulty}\}) = 1$, and $m(\{\text{faulty}\}) = m(\{\sim \text{faulty}\}) = m(\emptyset) = 0$. The former (i.e., $m(\{\text{faulty}, \sim \text{faulty}\}) = 1$) deals with a state of ignorance since the hypothesis set includes all possible hypotheses and therefore, its truth is believed to be certain.

For every mass function, there are associated functions of *belief* and *plausibility*. The degree of belief on A , $\text{bel}(A)$ is defined to be $\sum_{X \subseteq A} m(X)$ and the plausibility of A , $\text{pl}(A)$ is $1 - \text{bel}(\sim A) = \sum_{X \cap A \neq \emptyset} m(X)$. For example, $\text{bel}(\{\text{faulty}\}) = m(\{\text{faulty}\}) + m(\emptyset) = m(\{\text{faulty}\})$. In general, $\text{bel}(A) = m(A)$ for any singleton set $A \subseteq \mathcal{U}$ and in such a case the computation of bel is greatly reduced. However, $\text{bel}(A)$ is not necessary the same as $m(A)$ when A is not a singleton set. Thus, m , bel and pl can be derived from one another. It can be shown that the interval $[\text{bel}(A), \text{pl}(A)]$ contains the probability of A in the classic sense (see [10]). Thus, *belief*

and *probability* are different measures. In this paper, we use the terms *likelihood* and *belief* synonymously.

A mass function can be combined using various rules including the popular *Dempster's Rule of Combination*, which is a generalization of the Bayes rule. For $X, A, B \subseteq \mathcal{U}$, a combination rule of mass functions m_1 and m_2 , denoted by $m_1 \oplus m_2$ (or $m_{1,2}$) is defined as the following:

$$m_{1,2}(X) = m_1 \oplus m_2(X) = \frac{\sum_{A \cap B = X} m_1(A)m_2(B)}{1 - K}, \quad (6)$$

where $K = \sum_{A \cap B = \emptyset} m_1(A)m_2(B)$ and $m_1 \oplus m_2(\emptyset) = 0$.

The combination rule can be applied in pairs repeatedly to obtain a combination of multiple mass functions. The above rule strongly emphasizes the agreement between multiple sources of evidence and ignores the disagreement by the use of a normalization factor.

3. Proposed Approach

By exploiting program spectra, the proposed approach to the way an automated fault localizer builds on the concepts of Dempster-Shafer theory will be described. Section 3.1 discusses the formulation of mass functions, which are our main contributions. Section 3.2 discusses some characteristics of different fault localizers and how the proposed approach can be extended. Section 3.3 illustrates the approach in a small example along with intuitive justifications.

3.1. Mass Functions and Combination Rule. Mass functions are essential elements in estimating the likelihood of the code statement being faulty based on evidences from the program spectra described in Section 2. For any statement j of a tested program, let $\mathcal{U}_j = \{f_j, \sim f_j\}$, where f_j represents the hypothesis that j is faulty and similarly, $\sim f_j$ for nonfaulty. For each test run, we are concerned with whether the test was successful or not and what code statements were executed during the test. There are two possibilities.

Case 1 (failed test). A failed test that involves the execution of the statement under investigation is evidence that supports that the statement is likely to be faulty. In such a case, the likelihood of the statement being nonfaulty is zero. On the other hand, its likelihood of being faulty can be estimated by a ratio of one over the total number of statements involved in this test run. We can formulate this formally as follows

Recall that in the program spectra, $r_{ij} = 1$ if test i involves execution of unit j of the program, and it is 0, otherwise. Thus, a total number of units executed in test run i can be represented by $\sum_j r_{ij}$. We now define m_i , the mass function of failed test i for all possible nonempty subsets of the hypotheses in $\mathcal{U}_j$ as follows:

$$m_i(\{\sim f_j\}) = 0, \quad (7)$$

$$m_i(\{f_j\}) = \alpha * \left(\frac{r_{ij}}{\sum_j r_{ij}} \right), \quad \text{where } 0 < \alpha \leq 1, \quad (8)$$

$$m_i(\mathcal{U}_j) = 1 - m_i(\{f_j\}). \quad (9)$$

The third equation is derived from the property that $m(\emptyset) = 0$ and $\sum_{A \subseteq \mathcal{U}} m(A) = 1$. Based on the second equation, it should be easy to see that the likelihood of a statement being faulty can only be influenced by the (failed) test that executes that statement. The parameter α is an adjusted value that represents the strength of the property of “failed test” in determining if statement j is faulty.

Case 2 (passed test). If a test involving the execution of the statement in question is successful, it is evidence that supports that this statement behaves correctly. Thus, the likelihood of it being faulty is zero. On the other hand, the likelihood of this statement being correct (i.e., nonfaulty) can be estimated by a ratio of one over the total number of statements involved in this test run. We now define m_i , the mass function of passed test i for all possible nonempty subsets of the hypotheses in $\mathcal{U}_j$ as follows:

$$m_i(\{f_j\}) = 0, \quad (10)$$

$$m_i(\{\sim f_j\}) = \beta * \left(\frac{r_{ij}}{\sum_j r_{ij}} \right) \quad \text{where } 0 < \beta \leq 1, \quad (11)$$

$$m_i(\mathcal{U}_j) = 1 - m_i(\{\sim f_j\}). \quad (12)$$

It should be easy to see that in this case the likelihood of a statement being correct can only be influenced by the (successful) test that executes it. Analogous to α , the parameter β is an adjusted value that represents the strength of the property of “passed test” in determining if statement j is not faulty.

In general, the appropriate values of the adjusted values α and β are determined empirically since they are likely to depend on the size of the program, the number of tests and the ratios of failed to passed tests (see an example in Section 3.3). The more precision the values of α and β are, the more likely we can discriminate faulty belief values among a large number of software units. In this paper, α and β are estimated conservatively to one and 0.0001, respectively, in order to yield a sufficient power of discrimination for a very large program. The intuitive reason behind this is the fact that when a test fails, we can guarantee the existence of at least one faulty line. Thus, we should give very high strength to such evidence. This justifies α having the highest possible strength of one. However, when a test is successful, there is no guarantee that there is no faulty statement since the particular test may not have executed faulty statements and detected the faults. Thus, a successful test does not contribute to the belief of a statement being faulty as much as a failed test does. Nevertheless, when a statement is executed in a successful test, one may be inclined to believe that the statement is probably not likely to be faulty. As the number of such successful tests increases, we gain more confidence of such belief and thus, the successful test results have a contributing factor (although small) to the belief and cannot be ignored. In practice, the number of failed tests is typically much less than that of successful ones. Thus, each successful test should have less strength compared to that of each failed test. This explains why β takes a very small value that is as

close to zero as possible. We conjecture that the larger the size of the program and the smaller ratio of the number of failed tests to the number of successful tests would likely require much smaller β . However, to conclude such a statement requires further experiments.

Recall that a mass function of a singleton set hypothesis is the same as the degree of belief of the hypothesis. Applying one of the above two cases, a mass function is created for each of the supporting evidences (i.e., a test result, which either supports faulty or nonfaulty) of each program statement. Thus, the likelihood of a statement being faulty is estimated by combining the beliefs obtained from corresponding mass functions for each of the supporting pieces of evidence. To define the rule for combining mass functions, suppose that m_1 and m_2 are two distinct mass functions of a particular code statement i . Dempster's rule of combination can be applied as shown below. For readability, we omit i and replace $\{f_i\}$, $\{\sim f_i\}$ and $\mathcal{U}_i$ by f , $\sim f$ and $\mathcal{U}$, respectively.

$$\begin{aligned} m_{1,2}(f) &= \frac{(m_1(f)m_2(f) + m_1(f)m_2(\mathcal{U}) + m_1(\mathcal{U})m_2(f))}{1 - K}, \\ m_{1,2}(\sim f) &= \frac{(m_1(\sim f)m_2(\sim f) + m_1(\sim f)m_2(\mathcal{U}) + m_1(\mathcal{U})m_2(\sim f))}{1 - K}, \\ m_{1,2}(\mathcal{U}) &= \frac{m_1(\mathcal{U})m_2(\mathcal{U})}{1 - K}, \end{aligned} \quad (13)$$

where $K = m_1(f)m_2(\sim f) + m_1(\sim f)m_2(f)$.

This combination rule can be applied repeatedly pairwise until evidence from all test runs has been incorporated into the computation of the likelihood of each statement. Unlike other spectrum-based fault localizers discussed in Section 2.2, instead of ranking the lines based on the similarity coefficient values, our proposed approach ranks the lines based on the corresponding likelihood of them being faulty using the beliefs combined from all of the test evidence.

3.2. Characteristic Comparisons and Generalization. This section discusses and compares some characteristics of the proposed approach with other fault localizers: *Tarantula*, *Jaccard*, and *Ochiai*. Our approach is based on the principle of uncertainty reasoning, while the others are based on similarity coefficients, which share common characteristics. We now describe them below assuming that a code statement in question is given.

Suppose we classify test runs into those that *failed* and those that *executed* the code statement. The coefficient of each of the three localizers reaches its minimum value of zero when there is no test in both groups (i.e., $a_{11} = 0$). This means that there is no association between failed tests and tests that executed the code statement. On the other hand, the coefficient reflects a maximum association with a value one when all tests are in both groups. In other words, there is no failed test that did not execute the statement (i.e., $a_{01} = 0$) and no passed test that executed the statement (i.e., $a_{10} = 0$).

Unlike *Tarantula*, *Jaccard*, and *Ochiai* do not use a_{00} to compute the similarity coefficient. The denominator of the *Jaccard* coefficient represents the “sum” of all *failed* (i.e.,

$a_{11} + a_{01}$) and all *executed* tests (i.e., $a_{11} + a_{10}$) but not both (i.e., excluding a_{11}). On the other hand, the denominator of the *Ochiai* coefficient, which is derived from a geometric mean of a_{11}/failed and $a_{11}/\text{executed}$, is a square root of a “product” of *executed* and *failed* tests. Thus, the *Ochiai* coefficient amplifies the distinction between *failed* and *executed* tests more than the *Jaccard* coefficient. Therefore, the *Ochiai* coefficient is expected to provide a better discriminator for locating fault units.

Recall that the similarity coefficient of *Tarantula* is $\%failed/(\%failed + \%passed)$. This seems reasonable. However, note that when $\%passed$ value is zero, the coefficient value is one regardless of the $\%failed$ value. Suppose that the two code statements both have zero $\%passed$ but one was executed in one out of 100 failed tests and the other was executed in 90 out of the 100 failed tests. The chance of the latter statement being faulty should be greater than the other but *Tarantula* would rank the two statements as equally likely to be faulty. This explains why *Ochiai* and *Jaccard* can outperform *Tarantula* as reported in [10].

Our approach uses the Dempster-Shafer Theory of Evidence to account for each test run to accumulatively support the hypothesis about the statement. This makes our approach applicable for online computation. As shown in (8), each failed test adds belief to the statement being faulty. Similarly, in (10), each passed test adds belief to the statement being correct. The belief is contributed to by a probability factor that depends on the number of times that the test executed the statement and an overall number of statements executed in that test. Thus, our reasoning is in a finer grain size than the others since it focuses on a specific test (i.e., the mass functions) and the contributing factors are not necessarily expressible in terms of a_{ij} 's as with other similarity coefficients. Thus, it would be interesting to compare the performance among these approaches.

The proposed approach can be generalized to accommodate evidence of new properties. That is, we can, respectively, generalize (8) and (10) to

$$\begin{aligned} m(\{f_i\}) &= \alpha * f, \quad \text{where } 0 < \alpha \leq 1, \\ m(\{\sim f_i\}) &= \beta * g, \quad \text{where } 0 < \beta \leq 1, \end{aligned} \quad (14)$$

where function f (function g) quantifies evidential support for statement i being faulty (correct). Thus, the proposed approach is easily extensible.

3.3. Illustrated Example. To demonstrate our approach, Algorithm 1 shows a small faulty program introduced in [10]. The program is supposed to sort a list of n rational numbers using a bubble sort algorithm. There is a total of five blocks (the last block corresponding to the body of the `RationalSort` function is not shown here). Block 4 is faulty since when we swap the order of the two rational numbers, their denominators (`den`) need to be swapped as well as the numerators (`num`).

The program spectra can be constructed after running six tests with various inputs as shown in Table 2. Tests 1, 2, and 6 are already sorted and so they result in no error. Test 3 is not sorted but because the input denominators are of the same

```

void rationalsort (
    int n, int* num, int* den)
{
    /* block 1 */
    int i, j, temp;
    for (i = n - 1; i >= 0; i--) {
        /* block 2 */
        for (j = 0; j < i; j++) {
            /* block 3 */
            if (RationalGT(num[j], den[j],
                num[j + 1], den[j + 1])) {
                /* block 4 */
                temp = num[j];
                num[j] = num[j + 1];
                num[j + 1] = temp;
            }
        }
    }
}

```

ALGORITHM 1: Example of a faulty program.

value, no error occurs. However, double errors occur during Test 4's execution and so errors go undetected and Test 4 is passed. Finally, Test 5 failed since it resulted in erroneous output of $\langle 1/1 \ 2/2 \ 4/3 \ 3/4 \rangle$.

In this small set of program spectra we use the adjusted strength α of value one and the adjusted strength β of value $1/6$ to reflect the ratio of the number of failed test to overall number of test runs. Applying our approach, since Test 1 is a passed test (i.e., error = 0), we compute the belief of each hypothesis using the mass functions (10), (11) and (12). For example, the beliefs of hypotheses related to Block 1 after Test 1 are

$$\begin{aligned}
 m_1(\{f_1\}) &= 0, \\
 m_1(\{\sim f_1\}) &= \beta * \left(\frac{r_{11}}{\sum_j r_{1j}} \right) = \left(\frac{1}{6} \right) * \left(\frac{1}{1} \right) = 0.167, \quad (15) \\
 m_1(\mathcal{U}_1) &= 1 - m_1(\{\sim f_1\}) = 0.833.
 \end{aligned}$$

Similarly, for Test 2, which is a passed test, we can apply the mass functions (10), (11) and (12) to compute the belief of each hypothesis. For example, the beliefs of hypotheses related to Block 1 after Test 2 are

$$\begin{aligned}
 m_2(\{f_1\}) &= 0, \\
 m_2(\{\sim f_1\}) &= \beta * \left(\frac{r_{21}}{\sum_j r_{2j}} \right) = \left(\frac{1}{6} \right) * \left(\frac{1}{2} \right) = 0.083, \\
 m_2(\mathcal{U}_1) &= 1 - m_2(\{\sim f_1\}) = 0.917.
 \end{aligned} \quad (16)$$

Now we can apply the Dempster's rule of combination to update the new beliefs as evidenced by the two tests. For simplicity, we omit the subscript representing software unit

of Block 1. Here $K = m_1(f)m_2(\sim f) + m_1(\sim f)m_2(f) = 0$ and we have

$$\begin{aligned}
 m_{1,2}(f) &= m_1(f)m_2(f) + m_1(f)m_2(\mathcal{U}) \\
 &\quad + m_1(\mathcal{U})m_2(f) = 0, \\
 m_{1,2}(\sim f) &= m_1(\sim f)m_2(\sim f) + m_1(\sim f)m_2(\mathcal{U}) \\
 &\quad + m_1(\mathcal{U})m_2(\sim f) = 0.236, \\
 m_{1,2}(\mathcal{U}) &= m_1(\mathcal{U})m_2(\mathcal{U}) = 0.764.
 \end{aligned} \quad (17)$$

Next consider Test 3, which is again a passed test. By applying the mass functions (10), (11), and (12), we obtain the following:

$$\begin{aligned}
 m_3(\{f_1\}) &= 0, \\
 m_3(\{\sim f_1\}) &= \beta * \left(\frac{r_{31}}{\sum_j r_{3j}} \right) = \left(\frac{1}{6} \right) * \left(\frac{1}{5} \right) = 0.333, \quad (18) \\
 m_3(\mathcal{U}_1) &= 1 - m_3(\{\sim f_1\}) = 0.967.
 \end{aligned}$$

By applying the Dempster's combination rule to $m_{1,2}$ and m_3 , we have the following:

$$\begin{aligned}
 m_{1,2,3}(f) &= m_{1,2}(f)m_3(f) + m_{1,2}(f)m_3(\mathcal{U}) \\
 &\quad + m_{1,2}(\mathcal{U})m_3(f) = 0, \\
 m_{1,2,3}(\sim f) &= m_{1,2}(\sim f)m_3(\sim f) + m_{1,2}(\sim f)m_3(\mathcal{U}) \\
 &\quad + m_{1,2}(\mathcal{U})m_3(\sim f) = 0.262, \\
 m_{1,2,3}(\mathcal{U}) &= m_{1,2}(\mathcal{U})m_3(\mathcal{U}) = 0.738.
 \end{aligned} \quad (19)$$

The above belief computation repeats until no more evidence from the test runs is to be considered. Thus, the belief of the hypothesis that Block 1 being faulty is calculated in accumulative fashion. For each block, the process continues for each test to accumulate the new beliefs of each hypothesis until all tests have been considered. Each new test run can be immediately integrated into the fault

TABLE 2: Program spectra and results.

Test input	Block 1	Block 2	Block 3	Block 4	Block 5	Error
(1){}	1	0	0	0	0	0
(2){1/4}	1	1	0	0	0	0
(3){2/1 1/1}	1	1	1	1	1	0
(4){4/1 2/2 0/1}	1	1	1	1	1	0
(5){3/1 2/2 4/3 1/4}	1	1	1	1	1	1
(6){1/4 1/3 1/2 1/1}	1	1	1	0	1	0
No. of matches	1	2	3	4	3	6
Fault Likelihood	0.15	0.17	0.18	0.19	0.18	

TABLE 3: Belief values using different adjusted strengths.

Unit	$\alpha = 0.9, \beta = 0.1$	$\alpha = 0.99, \beta = 0.01$	$\alpha = 1, \beta = 0.0001$
Block 1	0.149	0.1946	0.1999656
Block 2	0.163	0.1962	0.1999816
Block 3	0.170	0.1969	0.1999896
Block 4	0.174	0.1973	0.1999936
Block 5	0.170	0.1969	0.1999896

localization process. It is clear that our approach supports online computing. In this example, Test 5 is the only failed test to which we apply the mass functions (7), (8) and (9).

The final results obtained for the beliefs of each block being faulty are shown in the last row of Table 2. Ranking the beliefs obtained, Block 4 is identified to be the most likely faulty location. This is as expected. In fact, as shown in the second-to-bottom line of Table 2, Block 4 has the highest number of matched executions with the error results obtained from the six tests. Thus, the approach produces the result that is also in line with the concept of finding the block that corresponds most to the error results in the spectra.

As mentioned earlier that the adjusted strength values of α and β can be determined empirically for each specific program to obtain the best discriminative power among beliefs of each software unit. For example, Table 3 shows the belief values obtained for this small program using various adjusted strength values.

As shown in Table 3, all cases of the strength values are able to correctly identify Block 4 as a faulty unit. However, different strength values give different precision on the belief values, which can be critical for large-scale program spectra.

4. Evaluation and Comparison Study

To evaluate our approach, we compare the results of our approach with the top three state-of-the-art spectrum-based fault localizers using a popular benchmark data set.

4.1. Benchmark Data and Acquisition. The Siemens Program Suite [22], which has been widely used as a benchmark for testing fault localizer effectiveness [9–11]. The suite has seven programs with multiple versions that reflect real-world scenarios. Most versions contain a single fault except a few

TABLE 4: Siemens program suite.

Program Name	Versions omitted	No. versions	No. lines	No. test cases
print_tokens	4, 6	7	475	4130
print_tokens2	10	10	401	4115
replace	27, 32	32	512	5542
schedule	3, 5, 6, 7, 8, 9	9	292	2649
schedule2	9	10	297	2710
tcas	38	41	135	1608
tot_info	none	23	346	1052

containing two faults. However, since we typically locate one fault at a time, most studies including ours focus on methods for locating a single fault. The GNU Compiler Collection (GCC) 4.4.0 compiler was used to compile the programs and the GNU Coverage (GCov) extension was used to generate code coverage information to construct the program spectra.

For data preparation, we omit 13 out of a total of 132 faulty program versions due to inappropriate changes in the header file, lack of failure test results, and crash before GCov could produce a trace file. Thus, we use the remaining total of 119 versions in the experiment. Table 4 summarizes the information of the Siemens program suite including the program names, versions excluded and their corresponding number of (faulty) versions, executable lines of code, and test cases.

4.2. Evaluation Metric. Based on the standard method for evaluating fault localizers, we use *effectiveness* [18] as our evaluation metric. Effectiveness is defined to be the percentage of unexamined code (saved efforts) when identifying fault location. More precisely, suppose $P = \langle p_1, p_2, \dots, p_n \rangle$ is a list of program statements (units, blocks) ranking in a descending order of similarity coefficient or likelihood values. The *effectiveness*, which signifies the reduction of the amount of code examined to locate a single fault in the software, can be specified as

$$\text{effectiveness} = \left(\frac{1 - k}{n} \right) * 100\%, \quad (20)$$

where $k = \min_{1 \leq j \leq n} \{j \mid p_j \text{ in } P \text{ is actually a faulty line}\}$. In other words, k is the first code statement found in P that

actually is faulty. This is an optimistic measure that gives a maximum amount of unexamined code or effectiveness.

Because ranking can result in different findings for fault location, an issue of how to define the effectiveness when multiple lines have the same value of similarity coefficient or belief, can be crucial. Taking the first actual faulty statement found would be too optimistic and taking the last one found would be too pessimistic. Instead, Ali et al. [11] has proposed the *midline* adjustment, which appears to be a practical compromise. The midline takes the average rank of all the statements that have the same measure, which gives

$$\text{midline effectiveness} = \left(\frac{1 - \tau}{n} \right) * 100\%, \quad (21)$$

where $\tau = (k + m - 1)/2$, where k is defined above and $m = \max_{1 \leq j \leq n} \{j \mid p_j \text{ in } P \ \& \ p_j\text{'s measure} = p_k\text{'s measure}\}$.

Our experiments employ the *effectiveness* metric in both conventional and adjusted midline measures of effectiveness as described above to compare the performance of our approach with those of other three fault localizers, namely, *Tarantula*, *Jaccard*, and *Ochiai*.

4.3. Experimental Results. We implemented the four methods in C++. The experimental results show that in a total of 119 versions, our approach gives better or the same *effectiveness* as those of the other three 100% of the time. In fact, in about 40% of all cases, our results have higher effectiveness than those of the rest; specifically they are higher than those of *Tarantula*, *Jaccard*, and *Ochiai* in 61, 59, and 48 versions, respectively.

Figure 1 shows the results obtained for midline effectiveness, which is less optimistic than the conventional effectiveness. By ranking program versions based on their corresponding effectiveness, Figure 1 compares the average percentages, over all 119 versions, of midline effectiveness obtained by our approach to those of the others. For easy visualization, because *Tarantula* has the lowest performance in all versions, Figure 1 displays the resulting average midline effectiveness of all versions performed by *Tarantula* in increasing order.

As shown in Figure 1, our approach results in slightly higher average midline effectiveness than those of *Tarantula*, *Jaccard*, and *Ochiai*. In particular, compared with *Tarantula*, our approach approximately shows up to an average of 3% increase in the midline effectiveness. Thus, the proposed approach is competitive with top performing fault localizers.

To see how the midline effectiveness differs from the conventional “optimistic” effectiveness, we compare the results obtained from each metric. Figure 2 shows the average percentages of both types of effectiveness obtained by our approach in each of the programs in the Siemens set. As expected, the midline effectiveness gives slightly less optimistic effectiveness than those of the conventional ones.

Table 5 compares the average effectiveness of overall versions for each approach (in percentages). The numbers after “ $\pm$ ” represent variances. The conventional effectiveness is higher than that of the midline effectiveness. All methods perform competitively with no more than 0.5%

FIGURE 1: Comparison of average midline effectiveness.

TABLE 5: Average effectiveness in percentages.

Fault Localizer	% Avg. effectiveness	% Avg. midline effectiveness
Tarantula	88.10 $\pm$ 0.094	86.04 $\pm$ 0.088
Jaccard	88.14 $\pm$ 0.094	86.07 $\pm$ 0.088
Ochiai	88.34 $\pm$ 0.093	86.30 $\pm$ 0.087
Ours	88.47 $\pm$ 0.092	86.53 $\pm$ 0.086

difference of the effectiveness and all perform well with over 88% and 86% on the average of the conventional and midline effectiveness, respectively. As shown in Table 5, the proposed method gives the highest average percentage of effectiveness of both types, followed in order by *Ochiai*, *Jaccard*, and *Tarantula*. The proposed method also shows the least variance, so it performs consistently. However, the differences are marginal.

5. Impacts of Program Spectra

This section studies the impacts of program spectra on the effectiveness of fault localization. In previous sections, although our approach shows promising and competitive results on the Siemens benchmark data, one may question whether it performs well in general, regardless of what program spectra we use. It raises the issue whether the spectra quality or quantity have any impact on the effectiveness of the proposed fault localizer. To better understand how the proposed approach performs under different conditions of program spectra, we further perform our experiments using the same concepts and methodologies introduced in [10]. The effectiveness measure in this section refers to the midline effectiveness described earlier.

5.1. Quality Measure and Impacts. In a collection of program spectra as defined in Section 2, a software unit whose column vector exactly matches with the error vector has more assurance of being faulty. This is because the program fails (represented by a value one in a corresponding error vector entry) if and only if it is executed (represented by a value one in a corresponding entry of the software unit vector).

FIGURE 2: Comparison of two effectiveness measures.

Locating faults in such a case can be easily achieved. However, in practice inexact matches are quite common because a program may not fail even when a faulty unit is executed. It is possible to have a faulty unit executed in a successful test run since the run may not involve a condition that introduces errors or the errors may not propagate far enough to result in a program failure. Unfortunately, the ratio between these two cases is often unknown making fault localization more complex and difficult. In fact, the higher the number of successful test runs that involved the execution of software unit i is, the less confident we are that unit i is faulty. In other words, $a_{10}(i)$ is inversely proportional to the support in locating software fault at unit i . On the other hand, as pointed out earlier, fault localization of a software unit is easy if the program fails only whenever it is executed. Thus, the number of failed test runs that involved the execution of software unit i provides support in identifying unit i as faulty. That is, $a_{11}(i)$ is proportional to the support of fault localization at unit i . Based on the above arguments, we define a *support* for locating fault at software unit i , denoted by $\text{support}(i)$, as follows:

$$\text{support}(i) = \frac{a_{11}(i)}{a_{11}(i) + a_{10}(i)}. \quad (22)$$

In other words, the percentage of support for locating a faulty software unit is quantified by the ratio of the number of its executions in failed test runs to the total number of its executions. The support measure can be computed using information obtained from the set of program spectra used for locating faults. The higher the support value is, the more confident it is to locate the fault. Therefore, the support measure is an indicator of the quality of the program spectra in facilitating fault localization. When a fault location is known, we can compute the quality of support for locating the fault.

Each faulty version of a program in the benchmark set has an inherent *support*, whose value depends on various attributes including the type of fault and running environments of the test cases as reflected on the resulting program spectra. In the Siemens set, the support values of different faulty units range from 1.4% to 20.3%. We want to see how the quality of program spectra, as indicated

FIGURE 3: Quality impacts on effectiveness.

by varying support values, impacts the effectiveness of the proposed fault localization approach. To do this, for each given faulty location, a different support value can be obtained by excluding runs that contribute either to a_{11} (to obtain a series of smaller support values) or a_{10} (to obtain a series of larger support values). The passed or failed runs for exclusion are randomly selected from the available set when there are too many choices. In this experiment, each controlled support value is obtained by using the maximum number of relevant runs. For example, suppose a faulty unit is executed in 10 failed and 30 passed runs. A support value of 25% would be generated from all of the available runs instead of say one passed and three failed runs. As observed in [10], although there are alternatives (e.g., setting failed runs to passed or vice versa) for controlling the support values, excluding runs is a preferred method because it maintains integrity and consistency of the data. Using the exclusion method, we obtained support values ranging from 1% to 100% (in a 10% increment).

Figure 3 shows the % of average effectiveness over all 120 faulty program versions for each approach under the study and for the support values ranging from 1% to 100%. As shown in Figure 3, the proposed approach outperformed the rest regardless of the support quality of the program spectra. As support value increases, *Jaccard's* effectiveness moves toward that of the *Ochiai*, which consistently performed better than *Tarantula*. Each of the approaches provides the effectiveness of at least about 84% even at only 1% of quality support. Comparing these values with 100% of quality support from program spectra, the effectiveness obtained by our approach, *Ochiai*, *Jaccard*, and *Tarantula* increases by 3.29%, 3.35%, 3.42%, and 2.56%, respectively. The improvement of our proposed approach over *Ochiai*, the second best performing approach, ranges from 0.19% to 0.58%. The small improvements are obtained for higher quality support of 90%–100% and large improvement for lower support values of 20%–50%. This implies that the proposed approach performs more consistently and is more robust to the support quality of the program spectra than the *Ochiai* approach. As expected, the effectiveness of each

FIGURE 4: Quantity impact on the proposed approach.

approach increases as the quality of the program spectra to support fault localization increases. However, for the most part, the impact does not appear to be significant.

5.2. Quantity Impacts. This section investigates the quantity impact of program spectra using our approach. We evaluate the effectiveness by varying the number of passed and failed test runs involved in fault localization across the Siemens benchmark set. The spectra set contains a large number of runs ranging from 1052 to 5542 but it has a relatively small number of failed runs ranging from one to 518. Therefore, it is not possible to have all versions representing all possible cases of passed and failed runs. In our experiments, since interesting effects tend to appear in a small number of runs [10], we have focused on the range of one to 20 of passed and failed runs to maximize the utilization of the number of versions with both types of runs. Consequently, we used 86 of a total of 119 versions in the spectra set.

Figure 4 shows the effectiveness obtained from the proposed approach for the number of passed and failed tests varying from one to 20. Each entry represents an average of effectiveness (in percentages) across 86 selected versions. As shown in Figure 4, we can see that as the number of failed test runs increases, the average effectiveness increases. Thus, adding failed runs appears to improve the effectiveness. However, adding passed runs does not seem to change the average effectiveness obtained by our approach. We also found that the average effectiveness stabilized when the number of runs is around 18.

To gain understanding of our approach compared to others in this aspect, we performed a similar experiment with the *Ochiai* fault localizer. Figure 5 shows the average effectiveness obtained in percentages within the same range of 20 passed and failed tests. As shown in Figure 5, adding more failed runs appears to improve the effectiveness of fault localization when using the *Ochiai* technique. On the other hand, adding the number of passed runs can increase

FIGURE 5: Quantity impact on the *Ochiai* fault localizer.

or decrease the effectiveness especially when the number of passed tests runs toward 20 (from 16). These results obtained by the *Ochiai* localizer agree with those observed by Abreu et al. in [10]. Compared to the *Ochiai* localizer, our approach appears to be slightly more stable and less influenced by the size of passed runs in the benchmark set.

From our experiments on the quantity impact, it is evident that program spectra with more failed runs can slightly improve the effectiveness of fault localization using the proposed method. However, the spectra set with more passed test runs does not seem to have a great impact on its effectiveness. This is rather remarkable and distinct from the *Ochiai* method.

In general, a unit hit by a large number of passed runs should be less suspicious. Therefore, if such a unit were actually faulty, it would be difficult to detect. As a result, the effectiveness obtained by most spectrum-based fault localizers would be decreased. A similar argument applies when there is a large number of runs that can weaken the degree of similarity between the faulty unit and the errors detected from the tests. However, the total number of runs does not seem to impact the effectiveness of our approach. The reason that the proposed approach is not extremely sensitive to the number of passed runs (and thus, overall number of runs) is attributed to the strength parameters that already compensate the large difference between the number of passed and failed runs. Furthermore, performance of any fault localizer is likely to depend on individual programs and sets of program spectra collected for the analysis. Thus, by adjusting different values of strength, our approach can be easily customized to apply to the available data and enhance the fault localization effectiveness.

6. Related Work

Early automated software fault localization work can be viewed as a part of a software diagnostic system [3]. To locate

known faults, the system exploits knowledge about faults and their associated possible causes obtained from previous experience. For novel faults, the diagnostic system employs inferences based on the software model that contains program structure and functions along with heuristics to search for likely causes (and locations of faulty units). However, modeling the program behavior and functions is a major bottleneck in applying this technique in practice.

Agrawal et al. [23] introduced the fault localization approach based on *execution slices*, each of which refers to the set of a program's basic blocks executed by a test input. Weiser [4] introduced the concept of a program slice as an abstract used by programmers in locating bugs. Agrawal et al.'s approach assumes that the fault resides in the slice of a failed test and not in the slice of a successful test. By restricting the attention to the statements in the failed slice that do not appear in the successful slice, called *dice*, the fault is likely to be found. However, the technique does not fully exploit program spectra as it only uses a *single* failed (passed) test case's execution slice to locate faults [18]. Consequently, program statements that are executed by one passed test case and a different number of failed test cases would have an equal likelihood to be faulty if both were executed by the failed test case for the considered dice. Our approach, however, makes use of all the test cases in the test suite, both passed or failed, to estimate fault locations.

Recent spectrum-based fault localization techniques include *nearest neighbor queries* [5], *Tarantula* [13], *Jaccard* [12], and *Ochiai* [14]. These approaches use some measure of similarity to compare the units executed in a failed test to those of a passed test. The unit that results in the most similar behaviors to errors in all the test runs is likely to be faulty. Other fault localization techniques employ statistical [6, 7] and neural net models [24] to analyze source codes and program spectra, respectively. However, the performance of the former heavily relies on the quality of data, whereas the latter suffers from classic limitations such as local minima. Unlike these approaches, our approach is not similarity based or statistical based, but estimates the likelihoods or beliefs of units being faulty. The computation is based on a theory that is widely used for reasoning under uncertainty.

7. Discussion and Conclusions

Our approach is fundamentally different from existing methods. Although the results of the proposed approach on the benchmark data are marginally better than state-of-the-art approaches, it should be emphasized that the proposed approach provides useful aspects for software testing in practice. Its ability to do online computation to locate faults as soon as each test run is executed is particularly novel in that testing of large program can be performed more efficiently since it does not have to wait for all testing program spectral data to be completed. The intermediate results obtained can also influence the configuration or design of subsequent tests. This is useful for real-time systems where system configurations tend to be highly adaptable and it is hard to predict the program behaviors.

In summary, this paper presents a spectrum-based approach to fault localization using the Dempster-Shaffer theory of evidence. Other than its competitive performance to state-of-the-art techniques, our approach has several unique benefits. First, it is theoretically grounded and therefore it has a solid foundation for handling uncertainty in fault location. Second, it supports an online computation that allows the prediction of fault locations to be updated immediately as the execution of each test case is completed without having to wait for completion of a large enough set of program spectrum to be statistically valid. Such computation adapts well to real-time systems. Finally, the approach can be extended easily by adding new mass functions to represent additional evidence for use in the probability assignment of faulty hypotheses.

Future work includes more experiments to gain understanding of the characteristics of the proposed approach, for example, what types of program spectra on which the proposed approach would perform best or perform significantly better than the other three approaches, and how we can extend the approach so that it can deal with software with multiple faults. More experiments can be performed to see if different types of software units impact the results. These are among our ongoing and future research.

Acknowledgments

Thanks are due to Adam Jordan for his help on the experiments and to Phongphun Kijsanayothin for his helpful discussion and comments on earlier versions of this paper. The author would also like to thank the reviewers whose comments have helped improve the quality of the paper.

References

- [1] B. Hailpern and P. Santhanam, "Software debugging, testing, and verification," *IBM Systems Journal*, vol. 41, no. 1, pp. 4–12, 2002.
- [2] I. Vessey, "Expertise in debugging computer programs: an analysis of the content of verbal protocols," *IEEE Transactions on Systems, Man and Cybernetics*, vol. 16, no. 5, pp. 621–637, 1986.
- [3] R. Sedlmeyer, W. B. Thompson, and P. E. Johnson, "Diagnostic reasoning in software fault localization," in *Proceedings of the 8th International Joint Conference on Artificial Intelligence*, vol. 1, pp. 29–31, Morgan Kaufmann, San Francisco, Calif, USA, 1983.
- [4] M. Weiser, "Programmers use slices when debugging," *Communications of the ACM*, vol. 25, no. 7, pp. 446–452, 1982.
- [5] M. Renieres and S. P. Reiss, "Fault localization with nearest neighbor queries," in *Proceedings of the 18th IEEE International Conference on Automated Software Engineering*, pp. 30–39, Providence, RI, USA, 2003.
- [6] B. Liblit, M. Naik, A. X. Zheng, A. Aiken, and M. I. Jordan, "Scalable statistical bug isolation," in *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pp. 15–26, Association for Computing Machinery, New York, NY, USA, 2005.

- [7] C. Liu, L. Fei, X. Yan, J. Han, and S. P. Midkiff, "Statistical debugging: a hypothesis testing-based approach," *IEEE Transactions on Software Engineering*, vol. 32, no. 10, pp. 831–848, 2006.
- [8] T. Reps, T. Ball, M. Das, and J. Larus, "The use of program profiling for software maintenance with applications to the year 2000 problem," in *Proceedings of the 6th European Software Engineering Conference*, pp. 432–449, Springer, New York, NY, USA, 1997.
- [9] R. Abreu, P. Zoetewij, and A. van Gemund, "An evaluation of similarity coefficients for software fault localization," in *Proceedings of the 12th Pacific Rim International Symposium on Dependable Computing*, pp. 39–46, IEEE Computer Society Press, Riverside, Calif, USA, 2006.
- [10] R. Abreu, P. Zoetewij, and A. van Gemund, "On the Accuracy of Spectrum-based Fault Localization," in *Proceedings of the Testing: Academic and Industrial Conference Practice and Research Techniques (MUTATION '07)*, pp. 89–98, IEEE Computer Society Press, Washington, DC, USA, 2007.
- [11] S. Ali, J. H. Andrews, T. Dhandapani, and W. Wang, "Evaluating the accuracy of fault localization techniques," in *Proceedings of the IEEE/ACM International Conference on Automated Software Engineering*, pp. 76–87, IEEE Computer Society Press, Washington, DC, USA, 2009.
- [12] M. Y. Chen, E. Kiciman, E. Fratkin, A. Fox, and E. Brewer, "Pinpoint: problem determination in large, dynamic internet services," in *Proceedings of the International Conference on Dependable Systems and Networks (DSN '02)*, pp. 595–604, Berkeley, Calif, USA, 2002.
- [13] J. Jones, M. J. Harrold, and J. Stasko, "Visualization of test information to assist fault localization," in *Proceedings of the 24th International Conference on Software Engineering*, pp. 467–477, Association for Computing Machinery, New York, NY, USA, 2002.
- [14] A. Meyer, A. Garcia, A. Souza, and C. Souza Jr., "Comparison of similarity coefficients used for cluster analysis with dominant markers in maize (*Zea mays* L)," *Genetics and Molecular Biology*, vol. 27, no. 1, pp. 83–91, 2004.
- [15] S. Nessa, M. Abedin, E. W. Wong, L. Khan, and Y. Qi, "Software fault localization using N-gram analysis," in *Proceedings of the 3rd International Conference on Wireless Algorithms, Systems, and Applications*, pp. 548–559, Springer, Berlin, Germany, 2008.
- [16] V. Dallmeier, C. Lindig, and A. Zeller, "Lightweight defect localization for Java," in *Proceedings of the 19th European Conference on Object-Oriented Programming*, vol. 3568 of *Lecture Notes in Computer Science*, pp. 528–550, Springer, Glasgow, UK, 2005.
- [17] G. Shafer, *A Mathematical Theory of Evidence*, Princeton University Press, Princeton, NJ, USA, 1976.
- [18] J. Jones and M. J. Harrold, "Empirical evaluation of the tarantula automatic fault-localization technique," in *Proceedings of the 20th IEEE/ACM International Conference on Automated Software Engineering*, pp. 273–282, Association for Computing Machinery, New York, NY, USA, 2005.
- [19] P. Jaccard, "Étude comparative de la distribution florale dans une portion des Alpes et des Jura," *Bulletin de la Société Vaudoise des Sciences Naturelles*, vol. 37, pp. 547–579, 1901.
- [20] A. Ochiai, "Zoogeographic studies on the soleoid fishes found in Japan and its neighbouring regions," *Bulletin of the Japanese Society for the Science of Fish*, vol. 22, pp. 526–530, 1957.
- [21] J. Pearl, *Causality: Models, Reasoning, and Inference*, Cambridge University Press, 2000.
- [22] M. Hutchins, H. Foster, T. Goradia, and T. Ostrand, "Experiments of the effectiveness of dataflow- and control flow-based test adequacy criteria," in *Proceedings of the 6th International Conference on Software Engineering*, pp. 191–200, IEEE Computer Society Press, Sorrento, Italy, 1994.
- [23] H. Agrawal, J. R. Horgan, S. London, and W. E. Wong, "Fault localization using execution slices and dataflow tests," in *Proceedings of the 6th International Symposium on Software Reliability Engineering*, pp. 143–151, Toulouse, France, 1995.
- [24] W. Wong and Y. Qi, "BP neural network-based effective fault localization," *International Journal of Software Engineering and Knowledge Engineering*, vol. 19, no. 4, pp. 573–597, 2009.