Exploiting Count Spectra for Bayesian Fault Localization

Rui Abreu[‡] Alberto Gonzalez-Sanchez[†] Arjan J.C. van Gemund[†]

[‡]Department of Informatics Engineering
Faculty of Engineering, University of Porto
Portugal
rui@computer.org

[†]Department of Software Technology
Delft University of Technology
The Netherlands
a.j.c.vangemund@tudelft.nl

ABSTRACT

Background: Automated diagnosis of software defects can drastically increase debugging efficiency, improving reliability and time-to-market. Current, low-cost, automatic fault diagnosis techniques, such as spectrum-based fault localization (SFL), merely use information on whether a component is involved in a passed/failed run or not. However, these approaches ignore information on component execution *frequency*, which can improve the accuracy of the diagnostic process. **Aim**: In this paper, we study the impact of exploiting component execution frequency on the diagnostic quality. **Method**: We present a reasoning-based SFL approach, dubbed Zoltar-C, that exploits not only component involvement but also their frequency, using an approximate, Bayesian approach to compute the probabilities of the diagnostic candidates. Zoltar-C is evaluated and compared to other well-known, low-cost techniques (such as Tarantula) using a set of programs available from the Software Infrastructure Repository. **Results**: Results show that, although theoretically Zoltar-C can be of added value, exploiting component frequency does not improve diagnostic accuracy on average. **Conclusions**: The major reason for this unexpected result is the highly biased sample of passing and failing tests provided with the programs under analysis. In particular, the ratio between passing and failing runs, which has a major impact on the probability computations, does not correspond to the false negative (failure) rates associated with the actually injected faults.

Categories and Subject Descriptors

D.2.5 [**Software Engineering**]: testing and debugging

General Terms

Algorithms, Experimentation

Keywords

Debugging, reasoning approach, component count spectra.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
PROMISE2010, Sep 12-13, 2010. Timisoara, Romania
Copyright 2010 ACM ISBN 978-1-4503-0404-7...$10.00.

$$N \text{ spectra} \overbrace{\begin{bmatrix} a_{11} & a_{12} & \dots & a_{1M} \\ a_{21} & a_{22} & \dots & a_{2M} \\ \vdots & \vdots & \ddots & \vdots \\ a_{N1} & a_{N2} & \dots & a_{NM} \end{bmatrix}}^{M \text{ components}} \overbrace{\begin{bmatrix} e_1 \\ e_2 \\ \vdots \\ e_N \end{bmatrix}}^{\text{errors}}$$

Figure 1: Input for SFL

1. INTRODUCTION

When software failures are observed, developers/testers need to find their root cause as quickly as possible. Automatic fault localization techniques can be of considerable help to perform such rather cumbersome task [4]. Spectrum-based fault localization (SFL) is amongst the best (semi-)automatic techniques for fault localization. Current spectrum-based approaches for fault localization take as input an activity matrix A, that stores whether a component was involved in the run (test case), and pass/fail information e per test case (see Figure 1). Each row in A is called a spectrum. Classical, statistical SFL approaches use (A, e) to statistically correlate software component activity with program failures [6, 17, 19, 22, 26].

Reasoning-based approaches to SFL has been shown to have better diagnostic performance than statistical approaches as they imply model-based reasoning techniques [5]. Internally reasoning approaches uses a Bayesian approach based on a failure model (aka ϵ policy) that allows to exploit all information in the matrix.

Statistical approaches (as well as current spectrum-based reasoning) only consider whether a component is involved or not. Effectively, they do not exploit all information in A, i.e., the integer value of a_{ij} is mapped to 0 or 1 (aka a *component hit spectrum*). Unlike statistical approaches, reasoning-based SFL can exploit a_{ij} information by extending the current failure model [5] to take into account the *number of times* a component is called in the test (aka *component count spectra*).

In this paper we study the effect of extending the failure model to accommodate the integer values of a_{ij} on the diagnostic performance. In particular,

- We define a new failure model that estimates the failure probability of a test given the a_{ij} of the components involved, and outline its use in our Bayesian approach used in Zoltar [16]; We dubbed this new reasoning module Zoltar-C;

- We assess the impact in diagnostic performance when exploiting the integer a_{ij} compared to the $0-1$ mapping considered thus far. In particular, we measure the diagnostic performance impact for a set of well-known, commonly used programs taken from the Software Infrastructure Repository.

Our results show that exploiting component frequency does not improve the diagnostic process on average. The reason for these unexpected, and disappointing results is the biased sample of passing and failing tests provided with the programs under analysis. For instance, most test suites only offer a very limited fraction of failing runs, which does not statistically agree with the execution frequencies of the defective components in combination with their false negative rates (the percentage of tests that fail when defective components are executed). Depending on whether the faults reside in components with high a_{ij} frequency, large diagnosis errors can occur, compared to the diagnosis based on hit spectra.

The paper is organized as follows. In the next section we introduce some basic concepts and terminology, and illustrate the fault localization technique based on reasoning over program spectra. In Section 3 we present our Zoltar-C approach to fault localization. In Section 4, the approach is evaluated using real software programs to assess the true capabilities of our technique. We compare Zoltar-C with related work in Section 5. In Section 6 we conclude and discuss future work.

2. PRELIMINARIES

In this section we introduce program spectra, and describe how they are used for diagnosing software faults. We also give an overview of related work in the automated debugging area. First we introduce the necessary terminology.

2.1 Terminology

As in [8], the following terminology is used throughout this paper.

- A *failure* is an event that occurs when delivered service deviates from correct service.
- An *error* is a system state that may cause a failure.
- A *fault* is the cause of an error in the system.

In this paper we apply this terminology to computer programs that transform an input file to an output file in a single run. Specifically in this setting, faults are *bugs* in the program code, and failures occur when the output for a given input deviates from the specified output for that input. One specific form of failure is abnormal termination of a program, for example because of a segmentation fault.

To illustrate these concepts, consider the C function in Figure 2. It is meant to sort, using the bubble sort algorithm, a sequence of `n` rational numbers whose numerators and denominators are stored in the parameters `num` and `den`, respectively. There is a fault (bug) in the swapping code of block 4: only the numerators of the rational numbers are swapped while the denominators are left in their original order. In this case, a failure occurs when `RationalSort` changes the contents of its argument arrays in such a way that the result is not a sorted version of the original. An error occurs after the code inside the conditional statement is executed, while `den[j]` $\neq$ `den[j+1]`. Such errors can be latent: if we apply `RationalSort` to the sequence $\langle \frac{4}{1}, \frac{2}{2}, \frac{0}{1} \rangle$, an error occurs after the first two numerators are swapped. However, this error is "canceled" by later swapping actions, and the sequence ends up being sorted correctly. Note that faults do not automatically lead to errors, not even latent ones: no error will occur if the sequence is already sorted, or if all denominators are equal.

The purpose of *diagnosis* is to locate faults. Diagnosis applied to computer programs is known as debugging. The automated methods that we study here have wider applicability, but in the context of this paper they fall in the category of automated debugging techniques.

Error detection is a prerequisite for diagnosis. We must know that something is wrong before we can try to locate the fault. Failures constitute a rudimentary form of error detection, but many errors remain latent and never lead to a failure. An example of a technique that increases the number of errors that can be detected is array bounds checking. Failure detection and array bounds checking are both examples of *generic* error detection mechanisms, that can be applied without detailed knowledge of a program [1]. Other examples of mechanisms in this category are the detection of NULL pointer handling, `malloc` problems, and deadlock detection in concurrent systems. Examples of program specific mechanisms are precondition and postcondition checking, and the use of assertions.

2.2 Program Spectra

A program spectrum [23] is a collection of data that provides a specific view on the dynamic behavior of software. Typically, this data is collected at run-time, and consist of a number of *counters* of specific events/components. For example, *block count spectra* count how often every block (so a block is the grain-size of a component in this specific situation) of code is executed during a run of a program. In this case, a block of code is a C language *statement*, where we do not distinguish between the individual statements of a *compound statement*, but where we do distinguish between the cases of a *switch* statement[1]. So in Figure 2, the three assignments inside the body of the conditional statement constitute a single block.

To illustrate the concept of a program spectrum, suppose that the function `RationalSort` of Figure 2 is called from the following `main` function, to sort the sequence $\langle \frac{2}{1}, \frac{3}{1}, \frac{4}{1}, \frac{1}{1} \rangle$, which it happens to do correctly.

```
int main()
{
    /* block 0 */
    int num[] = { 2, 3, 4, 1 };
    int den[] = { 1, 1, 1, 1 };

    RationalSort(4, num, den);
    return 0;
}
```

Running this program would result in the block count spectrum represented by the histogram in Figure 3. Blocks 0 and 1, the bodies of functions `main` and `RationalSort`,

[1]This is a slightly different notion than a *basic block*, which is a block of code that has no branch.

```
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;
          }
       }
    }
 }
```

Figure 2: A faulty C function for sorting rational numbers

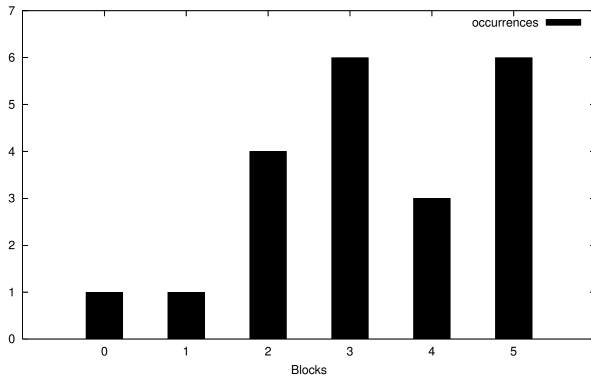

Figure 3: Block count spectrum

are both executed once. Blocks 2 and 3, the bodies of the two loops in `RationalSort` are executed four and six times, respectively. To sort the array in our example program we need to make three exchanges, and block 4, the `if` branch of the conditional statement, is executed three times. Block 5 has not been shown in Figure 2, but it represents the body of the `RationalGT` function. It is executed six times: once on every iteration of the inner loop.

If we are only interested in whether a block is executed or not, but not in the number of times it is executed, we can use binary flags instead of counters, and block count spectra revert to block hit spectra. Beside block count/hit spectra, many other forms of program spectra are used in practice. See [14] for an overview.

2.3 Fault Diagnosis

Program spectra can be used for fault diagnosis by comparing spectra for runs in which an error has been detected (usually called *failed* runs), to spectra for runs in which no error has been detected (usually called *passed* runs), and analyzing the differences. Using block spectra, this may identify those blocks (components) that are executed primarily in failed runs. These components are then also likely to contain the fault that causes the error. As we already pointed out, some form of error detection is needed to be able to make this classification of spectra, and failure detection provides a rudimentary form of error detection. We will now demonstrate the approach using our `RationalSort` example.

Suppose we apply `RationalSort` to the two sequences $I_1 = \langle \frac{1}{4}, \frac{1}{3}, \frac{1}{2}, \frac{1}{1} \rangle$ and $I_2 = \langle \frac{3}{1}, \frac{2}{2}, \frac{4}{3}, \frac{1}{4} \rangle$. The former sequence is already sorted, and the program will pass, but the latter sequence will result in a failure, which is a clear indication that an error has occurred. The block hit spectra for the two runs are as follows ('1' denotes component involvement and also that the run has failed).

	component						
input	0	1	2	3	4	5	error
I_1	1	1	1	1	0	1	0
I_2	1	1	1	1	1	1	1

The difference between the two hit spectra (correctly) identifies component 4 as the most likely location of the fault: while all other components are executed in both runs, component 4 only occurs in the run where the error is detected.

Of course, this example is contrived in many ways: the number of runs and components is small, no latent errors have occurred, no routine in the program has multiple call sites, etc. However, it serves to illustrate the basic principle.

2.3.1 Statistical Approach

Spectrum-based statistical fault localization essentially consists in identifying the component whose column vector (in A) resembles the error vector most (e). Such approach yields a ranked list of probable faulty components which should help the software tester to find the bugs quickly.

In the field of data clustering, resemblances between vectors of binary, nominally scaled data, such as the columns in our matrix of program hit spectra, are quantified by means of *similarity coefficients* (see, e.g., [15]). Many similarity coefficients exist. As an example, below are two different

similarity coefficients, namely the coefficient s_T, used in the Tarantula fault localization tool [17], and the Ochiai coefficient s_O, taken from the molecular biology domain and known to the best similarity coefficient for statistical fault localization based on program hit spectra [4, 6]:

$$s_T(j) = \frac{\frac{n_{11}(j)}{n_{11}(j)+n_{01}(j)}}{\frac{n_{11}(j)}{n_{11}(j)+n_{01}(j)} + \frac{n_{10}(j)}{n_{10}(j)+n_{00}(j)}} \quad (1)$$

$$s_O(j) = \frac{n_{11}(j)}{\sqrt{(n_{11}(j)+n_{01}(j)) \cdot (n_{11}(j)+n_{10}(j))}} \quad (2)$$

where $n_{11}(j)$ is the number of failed runs in which part j is involved, $n_{10}(j)$ is the number of passed runs in which part j is involved, $n_{01}(j)$ is the number of failed runs in which part j is not involved, and $n_{00}(j)$ is the number of passed runs in which part j is not involved, i.e., referring to Figure 1,

$$\begin{aligned} n_{00}(j) &= |\{i \mid a_{ij} = 0 \wedge e_i = 0\}| \\ n_{01}(j) &= |\{i \mid a_{ij} = 0 \wedge e_i = 1\}| \\ n_{10}(j) &= |\{i \mid a_{ij} = 1 \wedge e_i = 0\}| \\ n_{11}(j) &= |\{i \mid a_{ij} = 1 \wedge e_i = 1\}| \end{aligned}$$

Note that $n_{10}(j) + n_{11}(j)$ equals the number of runs in which part j is involved, and that $n_{10}(j) + n_{00}(j)$ and $n_{11}(j) + n_{01}(j)$ equal the number of passed and failed runs, respectively. The latter two numbers are equal for all j. Similarly, for all j, the four counters sum op to the number of runs N.

Under the assumption that a high similarity to the error vector indicates a high probability that the corresponding parts of the software cause the detected errors, the calculated similarity coefficients rank the parts of the program with respect to their likelihood of containing the faults.

To illustrate the approach, suppose that we apply the `RationalSort` function in Figure 2 to the input sequences shown in Table 1. The block hit spectra for these runs are shown in the central part of the table ('1' denotes a hit), where block 5 corresponds to the body of the `RationalGT` function, which has not been shown in Figure 2. The first, second, and sixth test cases are already sorted, and lead to passed runs. The third test case is not sorted, but the denominators in this sequence happen to be equal, hence no error occurs. For the forth test case an error occurs during its execution, but goes undetected. For the fifth test case the program fails, since the calculated result is $\langle \frac{1}{1}, \frac{2}{2}, \frac{4}{3}, \frac{3}{4} \rangle$ instead of $\langle \frac{1}{4}, \frac{2}{2}, \frac{4}{3}, \frac{3}{1} \rangle$, which is a clear indication that an error has occurred. For this data, the calculated similarity coefficients $s_{x\in\{T,O\}}(1), \ldots, s_{x\in\{T,O\}}(5)$ listed at the bottom of Table 1 (correctly) identify block 4 as the most likely location of the fault.

input	block 1	2	3	4	5	error
$\langle\ \rangle$	1	0	0	0	0	0
$\langle \frac{1}{4} \rangle$	1	1	0	0	0	0
$\langle \frac{2}{1}, \frac{1}{1} \rangle$	1	1	1	1	1	0
$\langle \frac{4}{1}, \frac{2}{2}, \frac{0}{1} \rangle$	1	1	1	1	1	0
$\langle \frac{3}{1}, \frac{2}{2}, \frac{4}{3}, \frac{1}{4} \rangle$	1	1	1	1	1	1
$\langle \frac{1}{4}, \frac{1}{3}, \frac{1}{2}, \frac{1}{1} \rangle$	1	1	1	0	1	0
s_T	0.50	0.56	0.63	0.71	0.63	
s_O	0.41	0.45	0.50	0.58	0.50	

Table 1: SFL applied on six runs of the `RationalSort` program

2.3.2 Reasoning Approach

As in real life many components may be likely explanations for observed failures, we need a mechanism to (1) derive the set of diagnosis candidate and (2) rank them according to their likelihood to be the true fault explanation. Below we describe the basic principles of spectrum-based reasoning, which were first introduced in [5].

Computing Diagnoses.

Our reasoning approach is inspired in model-based diagnosis (MBD). MBD is dependent on the existence of a model of the program. However, even if a model was available for each component (statement), only for the simplest of programs a program model could be extracted based on static dependence analysis. Unlike the MBD approaches, which statically deduce information from the program source [21], we use A as the *only*, dynamic source of information, from which both a model, and the input-output observations are derived. Apart from the fact that we exploit dynamic information, this approach also allows us to apply a generic component model, avoiding the need for detailed functional modeling, or relying, e.g., on invariants or pragmas for model information.

Abstracting from particular component behavior, each component c_j is modeled by the weak model

$$h_j \Rightarrow (x_j \Rightarrow y_j)$$

where h_j models the health state of c_j and x_j, y_j model its input and output variable value *correctness* (i.e., we abstract from actual variable values, in contrast to the earlier example). This weak model implies that a healthy component c_j translates a correct input x_j to a correct output y_j. However, a faulty component or a faulty input *may* lead to an erroneous output.

As each row in A specifies which components were involved, we interpret a row as a "run-time" model of the program as far as it was considered in that particular run. Consequently, A is interpreted as a sequence of typically different models of the program, each with its particular input and output correctness observation. The overall approach can be viewed as a sequential diagnosis that incrementally takes into account new program (and pass/fail) evidence with increasing N. A single row $A_{n,*}$ corresponds to the (sub)model

$$\begin{aligned} & h_m \Rightarrow (x_m \Rightarrow y_m), \text{ for } m \in S_n \\ & x_{s_i} = y_{s_{i-1}}, \text{ for } i \geq 2 \\ & x_{s_1} = \text{true} \\ & y_{s'} = \neg e_n \end{aligned}$$

where $S_n = \{m \in \{1, \ldots, M\} \mid a_{nm} = 1\}$ denotes the well-ordered set of component indices involved in computation n, s_i denotes the ith element in this ordering, (i.e., for $i \leq j, s_i \leq s_j$), and s' denotes its last element. The resulting

component chain logically reduces to

$$\bigwedge_{m \in S_n} h_m \Rightarrow \neg e_n$$

For example, consider the row ($M = 5$)

c_1	c_2	c_3	c_4	c_5	e
1	0	0	1	0	1

This corresponds to a model where components c_1, c_4 are involved. As the order of the component invocation is not given (and with respect to our above weak component model is irrelevant), we derive the model

$$\begin{aligned} &h_1 \Rightarrow (x_1 \Rightarrow y_1)\\ &h_4 \Rightarrow (x_4 \Rightarrow y_4)\\ &x_4 = y_1\\ &x_1 = \text{true}\\ &y_4 = \neg e_n \end{aligned}$$

In this chain the first component c_1 is assumed to have correct input ($x_1 = \text{true}$, typical of a proper test), its output feeds to the input of the next component c_4 ($x_4 = y_1$), whose output is measured in terms of e_n ($y_4 = \neg e_n$). This chain logically reduces to

$$h_1 \wedge h_4 \Rightarrow \text{false}$$

If this were a passing computation ($h_1 \wedge h_4 \Rightarrow \text{true}$) we could not infer anything (apart from the exoneration when it comes to probabilistically rank the diagnosis candidates as will be explained). However, as this run failed this yields

$$\neg h_1 \vee \neg h_4$$

which, in fact, is a *conflict* [12]. In summary, each failing run in A generates a conflict

$$\bigvee_{m \in S_n} \neg h_m$$

As in MBD, the conflicts are then subject to a hitting set algorithm that generates the diagnostic candidates [7, 11]. The minimal hitting set algorithm yields a set of valid diagnosis candidates. In this paper, we use a light-weight approach to compute the set of candidates given the conflicts called STACCATO (for interested readers, see [3] for details on the minimal hitting set algorithm).

Ranking Diagnoses.

Although the previous phase already excludes all those diagnoses that are irrelevant given the set of observed failures, the number of diagnosis candidates d_k is still typically large, and not all of them are equally probable to be the true fault explanation. Hence, the computation of a diagnosis candidate probabilities $\Pr(d_k)$ to establish a ranking is critical to the diagnostic performance of reasoning approaches. Although for each component the a priori fault probability $\Pr(\{j\})$ is typically dependent on code complexity, design, etc., we will simply assume $\Pr(\{j\}) = p$ ($p = 0.01$ in the context of this paper). Again, assuming components fail independently, the prior probability a particular diagnosis d_k is correct is given by $\Pr(d_k) = p^{|d_k|} \cdot (1-p)^{M-|d_k|}$. Similar to the incremental compilation of conflicts per run we compute the posterior probability for each candidate based on the pass/fail observation *obs* for each sequential run using Bayes' rule

$$\Pr(d_k|obs) = \frac{\Pr(obs|d_k)}{\Pr(obs)} \cdot \Pr(d_k)$$

where $\Pr(obs|d_k)$ is defined as

$$\Pr(obs|d_k) = \begin{cases} 0 & \text{if } d_k \text{ and } obs \text{ are inconsistent} \\ 1 & \text{if } d_k \text{ logically follows from } obs \\ \varepsilon & \text{if neither holds} \end{cases}$$

Due to the previous conflict-hitting set computation, the 0 case doesn't occur. Since the 1 case only applies to observations that can only occur for one particular fault, the ε case is the predominant one. Many policies exist for ε [7, 5, 11]. In this paper we compare our proposed approach against one of the best policies for software fault localization, which is defined as [7, 5]

$$\varepsilon = \begin{cases} g(d_k)^\eta & \text{if run passed} \\ 1 - g(d_k)^\eta & \text{if run failed} \end{cases} \tag{3}$$

where g denotes the probability that a defect, when executed, actually does not induce a program failure, and $\eta = |S_n|$ is the number of faulty components (according to d_k) involved (the rationale being that the more faulty components are involved, the more likely it is that the run will fail). The parameter g is estimated by [10]

$$g(d_k) = \frac{\sum_{n=1..N} [(\bigvee_{j \in d_k} a_{nj} \neq 0) \wedge e_n = 0]}{\sum_{n=1..N} [\bigvee_{j \in d_k} a_{nj} \neq 0]}$$

where $[\cdot]$ is Iverson's operator ($[\text{true}] = 1$, $[\text{false}] = 0$).

To illustrate how spectrum-based reasoning works, suppose that we run our example program with I_1 and I_2. In order to obtain the set of valid diagnosis candidates, the following model is derived from the (only) failed observation (see beginning of Section 2.3 for (A, e))

$$\begin{aligned} &h_0 \Rightarrow (x_0 \Rightarrow y_0)\\ &h_1 \Rightarrow (x_1 \Rightarrow y_1)\\ &h_2 \Rightarrow (x_2 \Rightarrow y_2)\\ &h_3 \Rightarrow (x_3 \Rightarrow y_3)\\ &h_4 \Rightarrow (x_4 \Rightarrow y_4)\\ &h_5 \Rightarrow (x_5 \Rightarrow y_5)\\ &x_5 = y_4\\ &x_4 = y_3\\ &x_3 = y_2\\ &x_2 = y_1\\ &x_1 = \text{true}\\ &y_4 = \neg e_n \end{aligned}$$

The model above logically reduces to

$$h_0 \wedge h_1 \wedge h_2 \wedge h_3 \wedge h_4 \wedge h_5 \Rightarrow \text{false}$$

And, therefore, the set of valid diagnosis candidates is $D = \{\{0\}, \{1\}, \{2\}, \{3\}, \{4\}, \{5\}\}$.

After the first phase - computing diagnoses - the set D of valid diagnosis candidates is ranked according to the

Bayesian update presented before, yielding the following diagnostic report

#	d_k	$\Pr(d_k)$
1	{4}	0.44
2	{0}	0.11
2	{1}	0.11
2	{2}	0.11
2	{3}	0.11
2	{5}	0.11

Hence, after running the program with this two test cases, spectrum-based reasoning would correctly pinpoint the developer to the (faulty) component. Note that for this specific scenario the ranking above would be the same for statistics-based techniques such as Tarantula and Ochiai. But, in general, reasoning yields better diagnostic results than approaches based on statistics.

3. ZOLTAR-C APPROACH

A particular problem with current approaches to spectrum-based reasoning for fault localization (such as the one discussed in the previous section) is that if components exhibit the same execution pattern, they will have exactly the same likelihood to be assigned a diagnosis. To illustrate this problem, suppose the spectra below which is obtained by running the `RationalSort` program with I_2 and $I_3 = \langle \frac{2}{1}, \frac{3}{1}, \frac{4}{1}, \frac{1}{1} \rangle$ only.

	component						
input	0	1	2	3	4	5	error
I_3	1	1	1	1	1	1	0
I_2	1	1	1	1	1	1	1

As spectrum-based reasoning approaches only exploit hit spectra, for this specific situation all six components rank with the same probability. Therefore, the fault localization approach would not bring any added value for the debugging problem, as in the worst case a developer would have to inspect all components (50% of the components would have to be inspected on average).

However, if count spectra would be considered instead of just hit spectra, a difference in the execution pattern would be immediately observed. The count spectra for the example above is as follows.

	component						
input	0	1	2	3	4	5	error
I_3	1	1	4	6	3	6	0
I_2	1	1	4	6	5	6	1

Hence, we can see that the faulty component is involved more times when the run fails. Therefore, this difference should be exploited to further indict that component as the most probable to be at fault.

We introduce a new epsilon policy, Zoltar-C, which exploits not only component involvement but also the number of times it was executed. The epsilon policy is given by

$$\varepsilon = \begin{cases} \prod_{j \in d_k \wedge a_{ij} \neq 0} g_j^{a_{ij}} & \text{if } e_i = 0 \\ 1 - \prod_{j \in d_k \wedge a_{ij} \neq 0} g_j^{a_{ij}} & \text{if } e_i = 1 \end{cases} \tag{4}$$

The reasoning behind this policy is that if a faulty component is involved multiple times in a run, then the probability that the run will fail is higher than if it only involved once. As an example, for $g_j = 0.90$ and a prior $p = 0.01$, the previous example Zoltar-C would yield the following ranking

#	d_k	$\Pr(d_k)$
1	{4}	$g_4^3 \cdot (1 - g_4^6) \cdot p = 0.27$
2	{3}	$g_3^6 \cdot (1 - g_3^6) \cdot p = 0.21$
2	{5}	$g_5^6 \cdot (1 - g_5^6) \cdot p = 0.21$
4	{2}	$g_2^4 \cdot (1 - g_2^4) \cdot p = 0.19$
5	{0}	$g_0^1 \cdot (1 - g_0^1) \cdot p = 0.07$
5	{1}	$g_1^1 \cdot (1 - g_1^1) \cdot p = 0.07$

For this specific example, by exploiting the count spectra available, Zoltar-C managed to move to the first position the faulty component, and therefore a developer would start by inspecting the real faulty component.

A problem with above policy is that g_j is both not known a priori and/or difficult to estimate. Although there are approaches for estimating the true intermittency rates g_j [5], in this paper we approximate that value using the previous approximation $g(d_k)$. Consequently, we redefine the policy as follows

$$\varepsilon = \begin{cases} g(d_k)^t & \text{if } e_i = 0 \\ 1 - g(d_k)^t & \text{if } e_i = 1 \end{cases} \tag{5}$$

where t is the number of combined frequency of the components in d_k, i.e., $t = \sum_{j \in d_k} a_{ij}$. Note that, in comparison with the previous ϵ-policy this one exploits the number of times components are involved in a run and not only the number of components in d_k involved in the run. So, $t \geq \eta$ always holds.

Finally, the workflow of our count spectra-based approach, Zoltar-C, is detailed in Algorithm 1. Like in other spectrum-based reasoning approaches [5], the set of diagnosis candidates D is computed from (A, e) using the light-weight algorithm STACCATO [3] (line 1). This ultra-low cost performance is achieved at the cost of completeness as solutions are truncated at 100 candidates. Nevertheless, experiments [3] have shown that no significant solution was ever missed. Subsequently, the probability of a given diagnosis candidate d_k being the true fault explanation is computed (lines 2 to 12). Finally, all probabilities are normalized so that they sum up to one (line 13), and the diagnostic report as well as the associated probabilities are returned to, e.g., the software developer.

4. EVALUATION

In this section, we evaluate the diagnostic capabilities and efficiency of Zoltar-C using real software programs. Before we present the experimental results, we describe the experimental setup and the performance metric.

4.1 Experimental Setup

For evaluating the performance of our approach we use the well-known Siemens benchmark set, as well as `gzip`, `sed` and `space` (obtained from the Software Infrastructure Repository[2] (SIR) [13]). The programs used in our experiments contain both seeded and real faults. Every single program has a correct version and a set of faulty versions of the same program. Although the faulty may span through multiple

[2]http://sir.unl.edu/

Algorithm 1 Diagnostic Algorithm: Zoltar-C

Inputs:
Activity matrix A
error vector e,

Output:
Diagnostic Report D
Diagnosis candidates Probabilities Pr

1 $D \leftarrow \textsc{Staccato}((A, e), 100)$
2 **for** all $d_k \in D$ **do**
3 $Pr[d_k] \leftarrow p^{|d_k|} \cdot (1-p)^{M-|d_k|}$
4 **for** all $i \in \{1, \ldots, N\}$ **do**
5 $t \leftarrow \sum_{j \in d_k} a_{ij}$
6 **if** $e_i = 0$ **then**
7 $Pr[d_k] \leftarrow Pr[d_k] \cdot g(d_k)^t$
8 **else**
9 $Pr[d_k] \leftarrow Pr[d_k] \cdot (1 - g(d_k)^t)$
10 **end if**
11 **end for**
12 **end for**
13 $(D, \Pr) \leftarrow \textsc{normalize}(D, \Pr)$
14 **return** $\textsc{sort}(D, \Pr)$

statements and/or functions, each faulty version contains exactly one fault. For each program a set of inputs is also provided, which were created with the intention to test full coverage. In particular, the `Space` package provides $1,000$ test suites that consist of a random selection of (on average) 150 test cases out of $13,585$ and guarantees that each branch of the program is exercised by at least 30 test cases. In our experiments, the test suite used is randomly chosen from the $1,000$ suites provided. Table 2 provides more information about the programs used in your experiments, where M corresponds to the number of lines of code (components in this context).

For our experiments, we have extended the subject programs with program versions where we can activate arbitrary combinations of *multiple* faults. For this purpose, we limit ourselves to a selection of 143 out of the 183 faults, based on criteria such as faults being attributable to a single line of code, to enable unambiguous evaluation.

As each program suite includes a correct version, we use the output of the correct version as reference. We characterize a run as failed if its output differs from the corresponding output of the correct version, and as passed otherwise.

4.2 Performance Metric

In the fault diagnosis research community rank-based [6, 17, 25] and dependency-based [19, 22] metrics have often been used. The former quantify the quality of a result based on the ranking position of the faulty component relative to all components, and is mainly used with techniques that rank components in a program. In contrast, dependency-based measures typically operate on the program dependence graph (PDG) and are mainly applied to evaluate techniques that either do not rank components (for example MBSD [21]) or do not rank all components of a program (such as SOBER [19]). Essentially, starting with the set of blamed components, dependencies between components are traversed in breadth-first order until the fault has been

Program	ϵ/Zoltar-C	Tarantula/Ochiai
`print_tokens`	4.2	0.37
`print_tokens2`	4.7	0.38
`replace`	6.2	0.51
`schedule`	2.5	0.24
`schedule2`	2.5	0.25
`tcas`	1.4	0.09
`tot_info`	1.2	0.08
`space`	7.4	0.15
`gzip`	6.2	0.19
`sed`	9.7	0.36

Table 4: Diagnosis cost for the single-fault subject programs (time in seconds)

reached. The quality of a diagnostic report is measured as the fraction of the PDG that is traversed. Both metrics quantify the percentage of a program that needs to be inspected in order to find the fault.

As spectrum-based fault localization techniques, such as the ones studied in this paper, create a ranking of statements in order of likelihood to be at fault, we can retrieve how many statements a software developer would have to inspect until he hits the faulty one. We define *wasted effort* (W) as the percentage of components that need be inspected when searching for the fault while traversing the ranking ($W = 0$ represents an ideal diagnosis: all faulted components are at the top of the ranking).

4.3 Performance Results

In this section we evaluate the diagnostic capabilities of Zoltar-C and compare it with other fault localization techniques.

Table 3 presents a summary of the diagnostic quality of the different techniques. The diagnostic quality is quantified in terms of wasted debugging effort W, as described in the previous section. The results show that, in general, Zoltar-C's average diagnostic quality is worse than techniques that use hit spectra. This disappointing performance has been observed across all programs as well as independent of the number of faults. Detailed box-and-whisker diagrams can be found at `http://www.st.ewi.tudelft.nl/~abreu/page.pdf`.

Table 4 summarizes the results of the time complexity study. We measure the time efficiency by conducting our experiments on a 2.3 GHz Intel Pentium-6 PC with 4 GB of memory. As expected, the less expensive techniques are the statistical techniques Tarantula and Ochiai. Zoltar-C and the ϵ policy for hit spectra (see Eq. (3) in Section 2.3.2) are equally complex. For example, for `sed`, the largest program in the set of programs analyzed by us, Zoltar-C/ϵ needs 9.7 seconds to produce the diagnostic report, where as Tarantula/Ochiai merely needs 0.36 seconds.

With respect to space complexity, statistical techniques, such as Tarantula and Ochiai, need two store the counters n_{11}, n_{10}, n_{01}, n_{00} for the similarity computation for all M components. Hence, the space complexity is O(M). Zoltar-C and ϵ also stores similar counters but per diagnosis candidate. Under the assumption that $|D|$ scales with M, these approaches have $O(M)$ space complexity.

4.4 Discussion

Program	Faulty Versions	M	N	Description
`print_tokens`	7	539	4,130	Lexical Analyzer
`print_tokens2`	10	489	4,115	Lexical Analyzer
`replace`	32	507	5,542	Pattern Recognition
`schedule`	9	397	2,650	Priority Scheduler
`schedule2`	10	299	2,710	Priority Scheduler
`tcas`	41	174	1,608	Altitude Separation
`tot_info`	23	398	1,052	Information Measure
`space`	38	9,564	150	ADL Interpreter
`gzip-1.3`	7	5,680	210	Data compression
`sed-4.1.5`	6	14,427	370	Textual manipulator

Table 2: The subject programs

		`print_tokens`			`print_tokens2`			`replace`			`schedule`			`schedule2`		
	C	1	2	4	1	2	4	1	2	5	1	2	5	1	2	5
	versions	4	6	1	10	43	100	23	100	100	7	20	11	9	35	91
MBD	ϵ	1.2	2.4	4.8	5.0	8.9	14.0	3.0	5.2	11.7	0.8	1.5	3.2	21.5	29.5	36.0
MBD	Zoltar-C	4.9	9.8	19.7	13.9	21.7	28.2	13.0	20.5	30.7	5.1	7.9	11.7	28.6	35.8	40.5
SFL	Ochiai	2.6	5.3	11.5	3.9	7.0	11.5	3.0	5.6	12.4	1.1	2.1	3.7	21.5	29.5	36.3
SFL	Tarantula	7.4	13.2	21.3	6.0	10.4	15.8	4.5	7.8	15.0	1.5	2.7	5.4	23.5	32.0	39.0

		`tcas`			`tot_info`			`space`			`gzip`			`sed`		
	C	1	2	5	1	2	5	1	2	5	1	2	5	1	2	5
	versions	30	100	100	19	100	100	28	100	100	7	21	21	5	10	1
MBD	ϵ	16.7	24.1	30.6	6.9	11.7	20.9	2.2	3.7	9.9	1.3	2.7	6.8	0.3	0.6	1.4
MBD	Zoltar-C	15.8	22.8	29.0	11.8	17.6	24.7	7.2	12.8	23.2	2.6	5.1	6.6	1.1	1.6	2.8
SFL	Ochiai	15.5	22.0	27.5	5.2	9.1	16.7	1.7	3.6	8.6	1.3	2.7	7.5	0.4	0.7	1.7
SFL	Tarantula	16.2	22.8	28.4	6.9	11.4	19.7	3.4	6.5	13.9	2.6	5.0	11.6	0.4	0.8	1.7

Table 3: Wasted effort W [%] on combinations of $C = 1 - 5$ faults for the subject programs

In this section we discuss in more depth why exploiting the count spectra leads to the above disappointing results. Two major factors influence the results. First, unlike in the hit spectrum case, the ϵ policy assumes that the failure probability $\Pr(e_i = 1)$ is governed by an or-model, i.e., any invocation of a faulty component that leads to a failure will lead to a program failure. While for different (uncorrelated) faults the or-model is reasonable (and has produced good results so far), for repeated invocations of the *same* faulty statement (e.g., involved in a loop) the or-model may not apply that well.

Apart from the above, there is a second, important cause for the observed results, which relates to the choice of test samples. Consider a case involving 2 components c_1, c_2, with c_2 at fault, and two tests, according to

c_1	c_2	e
1	10	0
1	10	1

Furthermore, suppose $g_1 = g_2 = g$. For the purpose of our exposition, for simplicity we assume a single fault. Our approach yields the posterior updates (non-normalized)

$$\begin{aligned} \Pr(c_1) &= (g_1^1)^1 \cdot (1 - g_1^1)^1 \cdot p = 0.25 \cdot p \\ \Pr(c_2) &= (g_2^{10})^1 \cdot (1 - g_2^{10})^1 \cdot p \approx 0.001 \cdot p \end{aligned}$$

Thus c_1 is computed to be much more probable to be at fault than c_2, in contrast to a hit spectrum interpretation (i.e., the a_{i2} entries were equal to 1) where both components would have equal probability.

The reason for the component with low a_{ij} entries to rank higher is the choice of test observations. In fact, given that c_2 is at fault, the probability of observing a same number of passing and failing tests (both 1 time in the above case) is extremely low. Given c_2 at fault, a proper sample of N tests should comprise $(1 - g_2^{10}) \cdot N$ failing tests versus $(g_2^{10}) \cdot N$ passing tests, i.e., an overwhelming majority of failing test cases. For $g_2 = 0.5$ this amounts to some 1,000 failing tests versus only 1 passing test, a distribution vastly different from the above, $N = 2$ test sample. In this case, the situating changes drastically, yielding the updates

$$\begin{aligned} \Pr(c_1) &= (g_1^1)^1 \cdot (1 - g_1^1)^{1000} \cdot p \approx 0.0000 \cdot p \\ \Pr(c_2) &= (g_2^{10})^1 \cdot (1 - g_2^{10})^{1000} \cdot p \approx 0.0003 \cdot p \end{aligned}$$

Indeed c_2 is now computed to be (much) more likely to be the fault.

While in hit spectra obtaining an unbiased sample of passing and failing tests is already an important requirement for proper diagnosis when using a Bayesian approach, the situation is much more critical when exploiting count spectra. The set of programs used in our experiments, unfortunately, does not offer an unbiased sample of passing and failing runs, as the set of runs (test inputs) is hand-picked to produce full coverage, rather than sampled from some representative input distribution. For instance, most test suites only offer a very limited fraction of failing runs. Depending on whether

the fault resides in a component with a high invocation frequency a_{ij} large diagnosis errors may occur, compared to the diagnosis based on hit spectra.

Apart from test sample quality, the way the g_j are approximated plays an equally important role, as estimation errors tend to get amplified for larger a_{ij} frequencies, again, leading to a loss of diagnostic precision. Rather than estimating the g_j on a biased sample of passing and failing tests, the g_j should preferably be determined outside of the diagnosis computation, based on mutation analysis.

4.5 Threats to Validity

The Siemens suite as well as several other utility programs have been used as subject programs in the study presented in this paper. All of them are either small or medium-sized programs. Further experiments on large programs may further strengthen the external validity of the overall conclusions. Currently, our tooling only supports C programs, and therefore we have not used programs in other programming languages. However, further investigation using other subject programs may help to generalize and strengthen our findings. Moreover, in our empirical study, we use the test suites provided by SIR, which were created to have full branch-coverage, but they may not represent all kinds of test cases in real-life situations.

Although we have thoroughly tested our tooling to ascertain correctness, another threat to validity is the correctness of our tooling, which is implemented using C.

Finally, another threat to validity is the type of faults we have at our disposal. Apart from the space program, all other programs have manually seeded faults. Therefore, further investigation with programs with real faults is needed to strengthen our confidence in our findings. We leave that for future work.

5. RELATED WORK

Depending on the amount of knowledge that is required about the system's internal component structure and behavior, the most predominant approaches to automatic fault localization can be classified as (1) statistical approaches or (2) reasoning approaches.

Statistical approaches use an abstraction of program traces (also known as program spectra), dynamically collected at runtime, to produce a list of likely candidates to be at fault. Well-known examples are Tarantula tool by Jones, Harrold, and Stasko [17], the Nearest Neighbor technique by Renieris and Reiss [22], the Sober tool by Lui, Yan, Fei, Han, and Midkiff [19], PPDG by Baah, Podgurski, and Harrold [9], CrossTab by Wong, Wei, Qi, and Zap [25], the Cooperative Bug Isolation by Liblit and his colleagues [18, 27], the Ochiai coefficient by Abreu, Zoeteweij, and Van Gemund [6], and the work of Wang, Cheung, Chan, and Zhang [24]. Although differing in the way they derive the statistical fault ranking, all techniques are based on measuring program hit spectra. In fact, due to the underlying diagnostic algorithms all these statistics-based approaches cannot exploit the extra information given by program count spectra.

Reasoning approaches combines a (automatically derived) static model of the expected behavior with a set of observations to compute the diagnostic report. In the work of Mayer and Stumptner [21] an overview of techniques to automatically generate program models from the source code is given, concluding that models generated by means of abstract interpretation [20] are the most accurate for model-based software debugging (MBSD). Reasoning approaches are much more complex than statistics-based approaches. In order to make reasoning approaches scale up to large programs, recently Abreu, Mayer, Stumptner, and Van Gemund proposed a framework, DEPUTO, combining model-based software debugging with statistics-based approach [2]. In recent work, we have also proposed a Bayesian (reasoning) approach, BARINEL, that solves the complexity problem in model-based debugging, taking a (hit) spectrum-based approach to MBSD, thus scaling to large programs [5].

The significant difference between the work proposed in this paper and the related work described above is that we propose a Bayesian approach that takes into account *count spectra*, i.e., we exploit the additional information on the number of times a component was involved in a given run instead of just considering component involvement. Neither of the above work considers execution frequencies.

6. CONCLUSIONS & FUTURE WORK

Current spectrum-based approaches to software fault localization, such as SFL, only take into account component involvement in a pass/fail test case, ignoring information on component execution frequency. In this paper we presented a reasoning-based SFL approach to fault localization, dubbed Zoltar-C, that do not only exploit component involvement but also frequency, which uses a Bayesian approach to compute their posteriors probabilities of being the most likely explanation for the observed failures.

We have evaluated and compared Zoltar-C to other renowned techniques such as Tarantula and Ochiai using a benchmark set of well-known software programs. Our results show that, although Zoltar-C can effectively aid developers in pinpointing the root cause of observed failures, it does not improve the diagnostic process on average for real programs. This unexpected result is due to two major factors. First, unlike in the hit spectrum case, the Bayesian probability update policy assumes that the failure probability is governed by an or-model, i.e., any of the invocations of component c_j leading to a failure will lead to a program failure. Second, the error is also due to the choice of test samples. The test suites provided with the subject programs under analysis have a highly biased sample of passing and failing test cases, which greatly influence the probability computations of the diagnosis candidates when execution frequencies are taken into account, and, by far, do not correspond to the false negative rates associated with the actually injected faults.

For future work we plan to theoretically study the impact of exploiting count spectra on the diagnostic performance of Zoltar-C, by using randomly generated matrices in order to vary several parameters such as number of runs, probability a component fails, and probability a component is involved in a run. In generating passing and failing runs we can correctly simulate the or-model, and also generate a non-biased sample of passing and failing runs, thus circumventing the current accuracy issues, and showing the theoretic potential of exploiting count spectra. This study, which is being performed at the moment, will increase our understanding of the results obtained for real software programs.

Dataset: availbale from the PROMISE repository.

7. REFERENCES

[1] R. Abreu, A. González, P. Zoeteweij, and A. van Gemund. Automatic software fault localization using generic program invariants. In *Proc. SAC'08*, Fortaleza, Brazil, March 2008. ACM Press.

[2] R. Abreu, W. Mayer, M. Stumptner, and A. J. C. van Gemund. Refining spectrum-based fault localization rankings. In *Proceedings of the Annual Symposium on Applied Computing (SAC'09)*.

[3] R. Abreu and A. J. C. van Gemund. A low-cost approximate minimal hitting set algorithm and its application to model-based diagnosis. In *Proceedings of Symposium on Abstraction, Reformulation, and Approximation (SARA'09)*.

[4] R. Abreu, P. Zoeteweij, R. Golsteijn, and A. J. C. van Gemund. A practical evaluation of spectrum-based fault localization. *Journal of Systems and Software*, 82(11):1780–1792, 2009.

[5] R. Abreu, P. Zoeteweij, and A. J. van Gemund. Spectrum-based multiple fault localization. In *Proceedings of the International Conference on Automated Software Engineering (ASE'09)*.

[6] R. Abreu, P. Zoeteweij, and A. J. C. van Gemund. On the accuracy of spectrum-based fault localization. In *Proceedings of The Testing: Academic and Industrial Conference - Practice and Research Techniques (TAIC PART'07)*.

[7] R. Abreu, P. Zoeteweij, and A. J. C. van Gemund. A new Bayesian approach to multiple intermittent fault diagnosis. In C. Boutilier, editor, *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI'09)*, pages 653 – 658, Pasadena, California, USA, 11 – 17 July 2009. AAAI Press.

[8] A. Avižienis, J.-C. Laprie, B. Randell, and C. E. Landwehr. Basic concepts and taxonomy of dependable and secure computing. *IEEE Trans. Dependable Sec. Comput.*, 1(1):11–33, 2004.

[9] G. K. Baah, A. Podgurski, and M. J. Harrold. The probabilistic program dependence graph and its application to fault diagnosis. In *Proceedings of International Symposium on Software Testing and Analysis (ISSTA'08)*.

[10] J. de Kleer. Diagnosing intermittent faults. In *Proceedings of International Workshop on Principles of Diagnosis (DX'07)*, May.

[11] J. de Kleer. Diagnosing multiple persistent and intermittent faults. In *Proceedings of the 21st International Joint Conference on Artificial Intelligence (IJCAI'09)*, pages 733–738, Pasadena, California, USA, 11 – 17 July 2009. AAAI Press.

[12] J. de Kleer, A. K. Mackworth, and R. Reiter. Characterizing diagnoses and systems. *Artificial Intelligence*, 56:197–222, 1992.

[13] H. Do, S. G. Elbaum, and G. Rothermel. Supporting controlled experimentation with testing techniques: An infrastructure and its potential impact. *Empirical Software Engineering: An International Journal*, 10(4):405–435, 2005.

[14] M. J. Harrold, G. Rothermel, R. Wu, and L. Yi. An empirical investigation of program spectra. In *Proceedings of International Workshop on Program Analysis for Software Tools and Engineering (PASTE'98)*.

[15] A. Jain and R. Dubes. *Algorithms for clustering data*. Prentice-Hall, Inc., 1988.

[16] T. Janssen, R. Abreu, and A. J. C. van Gemund. Zoltar: A toolset for automatic fault localization. In *Proceedings of the International Conference on Automated Software Engineering (ASE'09) - Tool Demonstrations*.

[17] J. A. Jones, M. J. Harrold, and J. Stasko. Visualization of test information to assist fault localization. In *Proceedings of International Conference on Software Engineering (ICSE'02)*.

[18] B. Liblit, M. Naik, A. X. Zheng, A. Aiken, and M. I. Jordan. Scalable statistical bug isolation. In *Proceedings of Conference on Programming Language Design and Implementation (PLDI'05)*.

[19] C. Liu, X. Yan, L. Fei, J. Han, and S. P. Midkiff. Sober: Statistical model-based bug localization. In *Proceedings of European Software Engineering Conference (ESEC) and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE-13)*.

[20] W. Mayer and M. Stumptner. Abstract interpretation of programs for model-based debugging. In *Proceedings of International Joint Conference on Artificial Intelligence (IJCAI'07)*.

[21] W. Mayer and M. Stumptner. Evaluating models for model-based debugging. In *Proceedings of International Conference on Automated Software Engineering (ASE'08)*.

[22] M. Renieris and S. P. Reiss. Fault localization with nearest neighbor queries. In *Proceedings of International Conference on Automated Software Engineering (ASE'03)*.

[23] 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 M. Jazayeri and H. Schauer, editors, *Proceedings of the Sixth European Software Engineering Conference (ESEC/FSE 97)*, volume 1301 of *LNCS*, pages 432–449. Springer–Verlag, 1997.

[24] X. Wang, S. Cheung, W. Chan, and Z. Zhang. Taming coincidental correctness: Coverage refinement with context patterns to improve fault localization. In *Proceedings of International Conference on Software Engineering (ICSE'09)*.

[25] W. Wong, T. Wei, Y. Qi, and L. Zhao. A crosstab-based statistical method for effective fault localization. In *Proceedings of International Conference on Software Testing and Verification (ICST'08)*.

[26] A. Zeller. Isolating cause-effect chains from computer programs. In *Proceedings of Symposium on the Foundations of Software Engineering (FSE'02)*.

[27] A. X. Zheng, M. I. Jordan, B. Liblit, M. Naik, and A. Aiken. Statistical debugging: Simultaneous identification of multiple bugs. In *Proceedings of International Conference on Machine Learning (ICML'06)*.