Exploiting Count Spectra for Bayesian Fault Localization

Rui Abreu‡
Faculty of Engineering, University of Porto
Portugal
rui@computer.org

Alberto Gonzalez-Sanchez†
Department of Informatics Engineering
Faculty of Engineering, University of Porto
Portugal
ru@computer.org

Arjan J.C. van Gemund†
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.

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 ε per test case (see Figure 1). Each row in A is called a spectrum. Classical, statistical SFL approaches use (A, ε) 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 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 $\langle \frac{1}{4}, \frac{3}{2}, \frac{1}{4} \rangle$, 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 $\langle \frac{1}{2}, \frac{3}{2}, \frac{1}{4} \rangle$, which it happens to do correctly.

```c
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

determine 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 $I_2 = \langle \frac{3}{4}, \frac{3}{2}, \frac{1}{2}, \frac{1}{1} \rangle$. 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).

<table>
<thead>
<tr>
<th>component</th>
<th>input</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>error</th>
</tr>
</thead>
<tbody>
<tr>
<td>$I_1$</td>
<td></td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>$I_2$</td>
<td></td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td></td>
</tr>
</tbody>
</table>

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 co-
ficient $s_O$, taken from the molecular biology domain and
known to the best similarity coefficient for statistical fault
localization based on program hit spectra [4, 6]:

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

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

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 Table 1,

$$n_{00}(j) = |\{ i \mid a_{ij} = 0 \land e_i = 0 \}|$$

$$n_{01}(j) = |\{ i \mid a_{ij} = 0 \land e_i = 1 \}|$$

$$n_{10}(j) = |\{ i \mid a_{ij} = 1 \land e_i = 0 \}|$$

$$n_{11}(j) = |\{ i \mid a_{ij} = 1 \land e_i = 1 \}|$$

Note that $n_{10}(j) + n_{11}(j)$ equals the number of runs in
which part $j$ is involved, and that $n_{10}(j) + n_{01}(j)$ 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 up 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 calcu-
lated 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 deno-
nimators 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 $\langle 4, 4, 4, 4, 4 \rangle$
instead of $\langle 4, 4, 4, 2, 4 \rangle$, which is a clear indication that an
error has occurred. For this data, the calculated similarity coeffi-
cients $s_x \in \{T, O\}^{(1), \ldots, s_x \in \{T, O\}}^{(5)}$ listed at the bottom of
Table 1 (correctly) identify block 4 as the most likely loca-
tion of the fault.

### 2.3.2 Reasoning Approach

As in real life many components may be likely explana-
tions for observed failures, we need a mechanism to (1) de-
rive the set of diagnosis candidate and (2) rank them ac-
cording 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].

<table>
<thead>
<tr>
<th>input</th>
<th>block</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>1</td>
</tr>
<tr>
<td>$\frac{1}{2}$</td>
<td>1</td>
</tr>
<tr>
<td>$\frac{1}{3}$</td>
<td>1</td>
</tr>
<tr>
<td>$\frac{1}{4}$</td>
<td>1</td>
</tr>
<tr>
<td>$\frac{1}{5}$</td>
<td>1</td>
</tr>
</tbody>
</table>

$s_T$ = $0.50, 0.56, 0.63, 0.71, 0.63$

$s_O$ = $0.41, 0.45, 0.50, 0.58, 0.50$

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

### Computing Diagnoses.

Our reasoning approach is inspired in model-based diag-
nosis (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 pro-
grams 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 in-
formation, 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 exam-
ple). This weak model implies that a healthy component $c_j$
translates a correct input $x_j$ to a correct output $y_j$. How-
ever, a faulty component or a faulty input may lead to an
erroneous output.

As each row in $A$ specifies which components were in-
volved, 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 dif-
ferent 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 $A_n, \ast$ corresponds to the (sub)model

$h_m \Rightarrow (x_m \Rightarrow y_m)$, for $m \in S_n$

$x_{s_i} = y_{s_{i-1}}$, for $i \geq 2$

$x_{s_1} = \text{true}$

$y_{s'} = \neg e_n$

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 $i^{th}$ 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 \((M = 5)\)

<table>
<thead>
<tr>
<th>c1</th>
<th>c2</th>
<th>c3</th>
<th>c4</th>
<th>c5</th>
<th>e</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

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

\[
\begin{align*}
   h_1 & \Rightarrow (x_1 \Rightarrow y_1) \\
   h_4 & \Rightarrow (x_4 \Rightarrow y_1) \\
   x_4 & = y_1 \\
   x_1 & = \text{true} \\
   y_4 & = \neg e_n 
\end{align*}
\]

In this chain the first component \(c_1\) is assumed to have correct input \((x_1 = \text{true}, \text{typical of a proper test})\), 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

\[
h_1 \land h_4 \Rightarrow \text{false}
\]

If this were a passing computation \((h_1 \land h_4 \Rightarrow \text{true})\) 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

\[
\neg h_1 \lor \neg h_4
\]

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 \(Pr(j) = 0\) \((p = 0.01\) 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

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

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 \(\varepsilon\) case is the predominant one. Many policies exist for \(\varepsilon\) \([7, 5, 11]\). In this paper we compare our proposed approach against one of the best policies for software fault localization, which is defined as \([7, 5]\)

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

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 \([10]\)

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

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 \((A, e)\))

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

The model above logically reduces to

\[
h_0 \land h_1 \land h_2 \land h_3 \land h_4 \land h_5 \Rightarrow \text{false}
\]

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 diagnostically report

<table>
<thead>
<tr>
<th></th>
<th>$d_k$</th>
<th>$\Pr(d_k)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>{4}</td>
<td>0.44</td>
</tr>
<tr>
<td>2</td>
<td>{0}</td>
<td>0.11</td>
</tr>
<tr>
<td>2</td>
<td>{1}</td>
<td>0.11</td>
</tr>
<tr>
<td>2</td>
<td>{2}</td>
<td>0.11</td>
</tr>
<tr>
<td>2</td>
<td>{3}</td>
<td>0.11</td>
</tr>
<tr>
<td>2</td>
<td>{5}</td>
<td>0.11</td>
</tr>
</tbody>
</table>

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. ZOultzar-C Approach

A particular policy 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 $I_3 = \langle \frac{2}{3}, \frac{2}{3}, \frac{1}{2}, \frac{1}{3} \rangle$ only.

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.

<table>
<thead>
<tr>
<th></th>
<th>$d_k$</th>
<th>$\Pr(d_k)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>{4}</td>
<td>0.44</td>
</tr>
<tr>
<td>2</td>
<td>{3}</td>
<td>0.27</td>
</tr>
<tr>
<td>3</td>
<td>{5}</td>
<td>0.21</td>
</tr>
<tr>
<td>4</td>
<td>{2}</td>
<td>0.19</td>
</tr>
<tr>
<td>5</td>
<td>{0}</td>
<td>0.07</td>
</tr>
<tr>
<td>6</td>
<td>{1}</td>
<td>0.07</td>
</tr>
</tbody>
</table>

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

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

(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

<table>
<thead>
<tr>
<th></th>
<th>$d_k$</th>
<th>$\Pr(d_k)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>{4}</td>
<td>0.44</td>
</tr>
<tr>
<td>2</td>
<td>{3}</td>
<td>0.27</td>
</tr>
<tr>
<td>3</td>
<td>{5}</td>
<td>0.21</td>
</tr>
<tr>
<td>4</td>
<td>{2}</td>
<td>0.19</td>
</tr>
<tr>
<td>5</td>
<td>{0}</td>
<td>0.07</td>
</tr>
<tr>
<td>6</td>
<td>{1}</td>
<td>0.07</td>
</tr>
</tbody>
</table>

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 Repository² (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

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 Repository² (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
Algorithm 1: Diagnostic Algorithm: Zoltar-C

**Inputs:**
- Activity matrix $A$
- error vector $e$,

**Output:**
- Diagnostic Report $D$
- Diagnosis candidates Probabilities $Pr$

1. $D \leftarrow \text{STACCATO}((A, e), 100)$
2. for all $d_k \in D$ do
   3. $Pr[d_k] \leftarrow p^{\lvert d_k \rvert} \cdot (1 - p)^{M - \lvert d_k \rvert}$
   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
5. end for
6. $(D, Pr) \leftarrow \text{NORMALIZE}(D, Pr)$
7. return sort($D, Pr$)

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

<table>
<thead>
<tr>
<th>Program</th>
<th>$\epsilon$/Zoltar-C</th>
<th>Tarantula/Ochiai</th>
</tr>
</thead>
<tbody>
<tr>
<td>print_tokens</td>
<td>4.2</td>
<td>0.37</td>
</tr>
<tr>
<td>print_tokens2</td>
<td>4.7</td>
<td>0.38</td>
</tr>
<tr>
<td>replace</td>
<td>6.2</td>
<td>0.51</td>
</tr>
<tr>
<td>schedule</td>
<td>2.5</td>
<td>0.24</td>
</tr>
<tr>
<td>schedule2</td>
<td>2.5</td>
<td>0.25</td>
</tr>
<tr>
<td>tcas</td>
<td>1.4</td>
<td>0.09</td>
</tr>
<tr>
<td>tot_info</td>
<td>1.2</td>
<td>0.08</td>
</tr>
<tr>
<td>space</td>
<td>7.4</td>
<td>0.15</td>
</tr>
<tr>
<td>gzip</td>
<td>6.2</td>
<td>0.19</td>
</tr>
<tr>
<td>sed</td>
<td>9.7</td>
<td>0.36</td>
</tr>
</tbody>
</table>

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](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 $\epsilon$ 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/$\epsilon$ needs 9.7 seconds to produce the diagnostic report, whereas 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 $\epsilon$ also store similar counters but per diagnosis candidate. Under the assumption that $|D|$ scales with $M$, these approaches have $O(M)$ space complexity.

4.4 Discussion
In this section we discuss in more depth why exploiting the count spectrum 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 \( Pr(c_1 = 1) \) 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

\[
\begin{array}{ccc}
\epsilon & 1 & 10 \\
\text{ZOLTAR-C} & 1.2 & 4.8 \\
\text{Ochiai} & 2.6 & 11.5 \\
\text{Tarantula} & 7.4 & 21.3 \\
\end{array}
\]

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)

\[
\begin{align*}
Pr(c_1) &= (g_1^1) \cdot (1 - g_1^1) \cdot p = 0.25 \cdot p \\
Pr(c_2) &= (g_2^1) \cdot (1 - g_2^1) \cdot p \approx 0.001 \cdot p 
\end{align*}
\]

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 \( a_{ij} \) 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 \((1 - g_2^1) \cdot N \) failing tests versus \((g_2^1) \cdot N \) 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

\[
\begin{align*}
Pr(c_1) &= (g_1^1) \cdot (1 - g_1^1) \cdot p \approx 0.000 \cdot p \\
Pr(c_2) &= (g_2^1) \cdot (1 - g_2^1) \cdot p \approx 0.0003 \cdot p 
\end{align*}
\]

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

<table>
<thead>
<tr>
<th>Program</th>
<th>Faulty Versions</th>
<th>M</th>
<th>N</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>print_tokens</td>
<td>7</td>
<td>539</td>
<td>1,330</td>
<td>Lexical Analyzer</td>
</tr>
<tr>
<td>print_tokens2</td>
<td>10</td>
<td>489</td>
<td>1,115</td>
<td>Lexical Analyzer</td>
</tr>
<tr>
<td>replace</td>
<td>32</td>
<td>507</td>
<td>5,542</td>
<td>Pattern Recognition</td>
</tr>
<tr>
<td>schedule</td>
<td>9</td>
<td>397</td>
<td>2,650</td>
<td>Priority Scheduler</td>
</tr>
<tr>
<td>schedule2</td>
<td>10</td>
<td>299</td>
<td>2,710</td>
<td>Priority Scheduler</td>
</tr>
<tr>
<td>tcas</td>
<td>41</td>
<td>174</td>
<td>1,608</td>
<td>Altitude Separation</td>
</tr>
<tr>
<td>tot_info</td>
<td>23</td>
<td>398</td>
<td>1,052</td>
<td>Information Measure</td>
</tr>
<tr>
<td>space</td>
<td>38</td>
<td>9,564</td>
<td>150</td>
<td>ADL Interpreter</td>
</tr>
<tr>
<td>gzip-1.3</td>
<td>7</td>
<td>5,680</td>
<td>210</td>
<td>Data compression</td>
</tr>
<tr>
<td>sed-4.1.5</td>
<td>6</td>
<td>14,427</td>
<td>370</td>
<td>Textual manipulator</td>
</tr>
</tbody>
</table>

Table 2: The subject programs

<table>
<thead>
<tr>
<th>C versions</th>
<th>print_tokens</th>
<th>print_tokens2</th>
<th>replace</th>
<th>schedule</th>
<th>schedule2</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>4</td>
<td>1</td>
<td>2</td>
<td>4</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
<td>6</td>
<td>1</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>3</td>
<td>4</td>
<td>8</td>
<td>1</td>
<td>2</td>
<td>4</td>
</tr>
<tr>
<td>4</td>
<td>5</td>
<td>10</td>
<td>1</td>
<td>2</td>
<td>5</td>
</tr>
<tr>
<td>5</td>
<td>6</td>
<td>12</td>
<td>1</td>
<td>2</td>
<td>6</td>
</tr>
<tr>
<td>6</td>
<td>7</td>
<td>14</td>
<td>1</td>
<td>2</td>
<td>7</td>
</tr>
<tr>
<td>7</td>
<td>8</td>
<td>16</td>
<td>1</td>
<td>2</td>
<td>8</td>
</tr>
<tr>
<td>8</td>
<td>9</td>
<td>18</td>
<td>1</td>
<td>2</td>
<td>9</td>
</tr>
<tr>
<td>9</td>
<td>10</td>
<td>20</td>
<td>1</td>
<td>2</td>
<td>10</td>
</tr>
</tbody>
</table>

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

\[
\begin{array}{cccccccccc}
\text{SFMBI} & \text{C} & \text{tcs} & \text{tot_info} & \text{space} & \text{gzip} & \text{sed} \\
\text{versions} & 1 & 2 & 5 & 1 & 2 & 5 & 1 & 2 & 5 \\
\text{ZOLTAR-C} & 10 & 10 & 100 & 10 & 100 & 10 & 100 & 10 & 100 \\
\text{Ochiai} & 10 & 10 & 100 & 10 & 100 & 10 & 100 & 10 & 100 \\
\text{Tarantula} & 10 & 10 & 100 & 10 & 100 & 10 & 100 & 10 & 100 \\
\end{array}
\]
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 \( g_{ij} \) 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 \( g_{ij} \) on a biased sample of passing and failing tests, the \( g_{ij} \) 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 combine 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 posterior 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 \( c \_i \) 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: available from the PROMISE repository.

7. REFERENCES


