On the Accuracy of Spectrum-based Fault Localization*

Rui Abreu **Peter Zoeteweij** **Arjan J.C. van Gemund**

Software Technology Department
Faculty of Electrical Engineering, Mathematics, and Computer Science
Delft University of Technology
P.O. Box 5031, NL-2600 GA Delft, The Netherlands
{r.f.abreu, p.zoeteweij, a.j.c.vangemund}@tudelft.nl

Abstract

Spectrum-based fault localization shortens the test-diagnose-repair cycle by reducing the debugging effort. As a light-weight automated diagnosis technique it can easily be integrated with existing testing schemes. However, as no model of the system is taken into account, its diagnostic accuracy is inherently limited. Using the Siemens Set benchmark, we investigate this diagnostic accuracy as a function of several parameters (such as quality and quantity of the program spectra collected during the execution of the system), some of which directly relate to test design. Our results indicate that the superior performance of a particular similarity coefficient, used to analyze the program spectra, is largely independent of test design. Furthermore, near-optimal diagnostic accuracy (exonerating about 80% of the blocks of code on average) is already obtained for low-quality error observations and limited numbers of test cases. The influence of the number of test cases is of primary importance for continuous (embedded) processing applications, where only limited observation horizons can be maintained.

Keywords: *Test data analysis, software fault diagnosis, program spectra.*

1 Introduction

Testing, debugging, and verification represent a major expenditure in the software development cycle [12], which is to a large extent due to the labor-intensive tasks of diagnosing the faults (bugs) that cause tests to fail. Because under typical market conditions, only those faults that affect the user most can be solved before the release deadline, the efficiency with which faults can be diagnosed and repaired directly influences software reliability. Automated diagnosis can help to improve this efficiency.

Diagnosis techniques are complementary to testing in two ways. First, for tests designed to verify correct behavior, they generate information on the root cause of test failures, focusing the subsequent tests that are required to expose this root cause. Second, for tests designed to expose specific potential root causes, the extra information generated by diagnosis techniques can help to further reduce the set of remaining possible explanations. Given its incremental nature (i.e., taking into account the results of an entire sequence of tests), automated diagnosis alleviates much of the work of selecting tests in the latter category, and can hence have a profound impact on the test-diagnose-repair cycle.

An important part of diagnosis and repair consist in localizing faults, and several tools for automated debugging and systems diagnosis implement an approach to fault localization based on an analysis of the differences in *program spectra* [20] for *passed* and *failed* runs. Passed runs are executions of a program that completed correctly, whereas failed runs are executions in which an error was detected. A program spectrum is an execution profile that indicates which parts of a program are active during a run. Fault localization entails identifying the part of the program whose activity correlates most with the detection of errors. Examples of tools that implement this approach are Pinpoint [6], which focuses on large, dynamic on-line transaction processing systems, Tarantula [17], which focuses on the analysis of C programs, and AMPLE [8], which focuses on object-oriented software (see Section 7 for a discussion).

Spectrum-based fault localization does not rely on

[*] This work has been carried out as part of the TRADER project under the responsibility of the Embedded Systems Institute. This project is partially supported by the Netherlands Ministry of Economic Affairs under the BSIK03021 program.

a model of the system under investigation. It can easily be integrated with existing testing procedures, and because of the relatively small overhead with respect to CPU time and memory requirements, it lends itself well for application within resource-constrained environments [24]. However, the efficiency of spectrum-based fault localization comes at the cost of a limited *diagnostic accuracy*. As an indication, in one of the experiments described in the present paper, on average 20% of a program still needs to be inspected after the diagnosis.

In spectrum-based fault localization, a *similarity coefficient* is used to rank potential fault locations. In earlier work [1], we obtained preliminary evidence that the Ochiai similarity coefficient, known from the biology domain, can improve diagnostic accuracy over eight other coefficients, including those used by the Pinpoint and Tarantula tools mentioned above. Extending as well as generalizing this previous result, in this paper we investigate the main factors that influence the accuracy of spectrum-based fault localization in a much wider setting. Apart from the influence of the similarity coefficient on diagnostic accuracy, we also study the influence of the quality and quantity of the (pass/fail) observations used in the analysis.

Quality of the observations relates to the classification of runs as passed or failed. Since most faults lead to errors only under specific input conditions, and as not all errors propagate to system failures, this parameter is relevant because error detection mechanisms are usually not ideal. Quantity of the observations relates to the number of passed and failed runs available for the diagnosis. If fault localization has to be performed at run-time, e.g., as a part of a recovery mechanism, one cannot wait to accumulate many observations to diagnose a potentially disastrous error until sufficient confidence is obtained. In addition, quality and quantity of the observations both relate to test coverage. Varying the observation context with respect to these two observational parameters allows a much more thorough investigation of the influence of similarity coefficients. Our study is based on the Siemens set [14] of benchmark faults (single fault locations).

The main contributions of our work are the following. We show that the Ochiai similarity coefficient consistently outperforms the other coefficients mentioned above. We establish this result across the entire quality space, and for varying numbers of runs involved. Furthermore, we show that near-optimum diagnostic accuracy (exonerating around 80% of all code on average) is already obtained for low-quality (ambiguous) error observations, while, in addition, only a few runs are required. In particular, maximum diagnostic performance is already reached at 6 failed runs on average. However, including up to 20 passed runs may improve but also degrade diagnostic performance, depending on the program and/or input data.

The remainder of this paper is organized as follows. In Section 2 we introduce some basic concepts and terminology, and explain the diagnosis technique in more detail. In Section 3 we describe our experimental setup. In Sections 4, 5, and 6 we describe the experiments on the similarity coefficient, and the quality and quantity of the observations, respectively. Related work is discussed in Section 7. We conclude, and discuss possible directions for future work in Section 8.

2 Preliminaries

In this section we introduce program spectra, and describe how they are used in software fault localization.

2.1 Failures, Errors, and Faults

As defined in [5], we use the following terminology. 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 simple 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.

To illustrate these concepts, consider the C function in Figure 1. 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 within the body of the `if` statement: 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 temporary, and do not automatically lead to failures. For example, 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.

Error detection is a prerequisite for the fault localization technique studied in this paper: we must know

```
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 1. A faulty C function for sorting rational numbers

that something is wrong before we can try to locate the responsible 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. Other examples 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 [20] is a collection of data that provides a specific view on the dynamic behavior of software. This data is collected at run-time, and typically consist of a number of counters or flags for the different parts of a program. Many different forms of program spectra exist, see [13] for an overview. In this paper we work with so-called block hit spectra.

A block hit spectrum contains a flag for every block of code in a program, that indicates whether or not that block was executed in a particular run. With a block of code we mean 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]. As an illustration, we have identified the blocks of code in Figure 1.

$$\begin{array}{ccc} & N\text{ parts} & \text{errors} \\ M\text{ spectra} & \begin{bmatrix} x_{11} & x_{12} & \dots & x_{1N} \\ x_{21} & x_{22} & \dots & x_{2N} \\ \vdots & \vdots & \ddots & \vdots \\ x_{M1} & x_{M2} & \dots & x_{MN} \end{bmatrix} & \begin{bmatrix} e_1 \\ e_2 \\ \vdots \\ e_M \end{bmatrix} \\ & \begin{matrix} s_1 & s_2 & \dots & s_N \end{matrix} & \end{array}$$

Figure 2. The ingredients of fault diagnosis

2.3 Fault Localization

The hit spectra of M runs constitute a binary matrix, whose columns correspond to N different parts (blocks in our case) of the program (see Figure 2). The information in which runs an error was detected constitutes another column vector, the error vector. This vector can be thought to represent a hypothetical part of the program that is responsible for all observed errors. Fault localization essentially consists in identifying the part whose column vector resembles the error vector most.

In the field of data clustering, resemblances between vectors of binary, nominally scaled data, such as the columns in our matrix of program spectra, are quantified by means of *similarity coefficients* (see, e.g., [15]). Many similarity coefficients exist. As an example, below are three different similarity coefficients, namely the Jaccard coefficient s_T, which is used by the Pinpoint tool [6], the coefficient used in the Tarantula fault localization tool [16], and the Ochiai coefficient s_O, used in the molecular biology domain:

$$s_J(j) = \frac{a_{11}(j)}{a_{11}(j) + a_{01}(j) + a_{10}(j)} \qquad (1)$$

$$s_T(j) = \frac{\frac{a_{11}(j)}{a_{11}(j)+a_{01}(j)}}{\frac{a_{11}(j)}{a_{11}(j)+a_{01}(j)} + \frac{a_{10}(j)}{a_{10}(j)+a_{00}(j)}} \qquad (2)$$

$$s_O(j) = \frac{a_{11}(j)}{\sqrt{(a_{11}(j) + a_{01}(j)) * (a_{11}(j) + a_{10}(j))}} \qquad (3)$$

where $a_{pq}(j) = |\{i \mid x_{ij} = p \land e_i = q\}|$, and $p, q \in \{0, 1\}$. Besides, $x_{ij} = p$ indicates whether block j was touched ($p = 1$) in the execution of run i or not ($p = 0$). Similarly, $e_i = q$ indicates whether a run i was faulty ($q = 1$) or not ($q = 0$).

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 to the input sequences

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

$I_1, \ldots, I_6$ (see below). The block hit spectra for these runs are as follows ('1' denotes a hit), where block 5 corresponds to the body of the `RationalGT` function, which has not been shown in Figure 1.

	block					
input	1	2	3	4	5	error
$I_1 = \langle\ \rangle$	1	0	0	0	0	0
$I_2 = \langle \frac{1}{4} \rangle$	1	1	0	0	0	0
$I_3 = \langle \frac{2}{1}, \frac{1}{1} \rangle$	1	1	1	1	1	0
$I_4 = \langle \frac{4}{1}, \frac{2}{2}, \frac{0}{1} \rangle$	1	1	1	1	1	0
$I_5 = \langle \frac{3}{1}, \frac{2}{2}, \frac{4}{3}, \frac{1}{4} \rangle$	1	1	1	1	1	1
$I_6 = \langle \frac{1}{4}, \frac{1}{3}, \frac{1}{2}, \frac{1}{1} \rangle$	1	1	1	0	1	0
s_J	.17	.20	.25	.33	.25	
s_T	.50	.56	.62	.71	.50	
s_O	.40	.44	.50	.58	.50	

I_1, I_2, and I_6 are already sorted, and lead to passed runs. I_3 is not sorted, but the denominators in this sequence happen to be equal, hence no error occurs. I_4 is the example from Section 2.1: an error occurs during its execution, but goes undetected. For I_5 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 $s_{x \in \{J,T,P\}}(1), \ldots, s_{x \in \{J,T,P\}}(5)$ (correctly) identify block 4 as the most likely location of the fault.

3 Experimental Setup

In this section we describe the benchmark set that we use in our experiments. We also detail how we extract the data of Figure 2, and define how we measure diagnostic accuracy.

3.1 Benchmark Set

In our study we worked with a widely-used set of test programs known as the *Siemens set* [14], which is composed of seven programs. Every single program has a correct version and a set of faulty versions of the same program. Each faulty version contains exactly one fault. However, the fault may span through multiple statements and/or functions. Each program also has a set of inputs that ensures full code coverage. Table 1 provides more information about the programs in the package (for more detailed information refer to [14]).

In our experiments we were not able to use all the programs provided by the Siemens set. Because we conduct our experiments using block hit spectra, we can not use programs which contain faults located outside a block, such as global variables initialization. Versions 4 and 6 of `print_tokens` contain such faults and were therefore discarded. Version 9 of `schedule2` and version 32 of `replace` were not considered in our experiments because no test case fails and therefore the existence of a fault was never revealed. Furthermore, as we are comparing ranking techniques, we decided to limit our experiment to single site faults. Hence, versions 12, and 21 of `replace`, versions 10, 11, 15, and 40 of `tcas`, version 7 of `schedule`, and version 1 of `print_tokens` were also discarded because the fault is extended to more than one site. In total, we discarded 12 versions out of 132 versions provided by the suite, using 120 versions in our experiments.

3.2 Data Acquisition

Collecting Spectra For obtaining block hit spectra we automatically instrumented the source code of every single program in the Siemens set using the parser generator `Front` [4], which is used in the development process within our industrial partner in the TRADER project [10]. A function call was inserted at the beginning of every block of code to log its execution (see [2] for details on the instrumentation process). Instrumentation overhead has been measured to be approximately 6% on average (with standard deviation of 5%). Moreover, the programs were compiled on a `Fedora Core release 4` system with `gcc-3.2`.

Error Detection As for each program the Siemens set includes a correct version, we use the output of the correct version of each program as error detection reference. We characterize a run as 'failed' if its output differs from the corresponding output of the correct version, and as 'passed' otherwise.

3.3 Evaluation Metric

As spectrum-based fault localization creates a ranking of blocks in order of likelihood to be at fault, we can retrieve how many blocks we still need to inspect until we hit the faulty block. If there are two or more blocks ranking with the same coefficient, we use the average ranking position for all the blocks.

Let $d \in \{1, \ldots, N\}$ be the index of the block that we know to contain the fault. For all $j \in \{1, \ldots, N\}$, let s_j denote the similarity coefficient calculated for block j. Then the ranking position is given by

$$\tau = \frac{|\{j | s_j > s_d\}| + |\{j | s_j \geq s_d\}| - 1}{2} \tag{4}$$

We define accuracy, or quality of the diagnosis as the effectiveness to pinpoint the faulty block. This metric represents the percentage of blocks that need not be considered when searching for the fault by traversing the ranking. It is defined as

$$q_d = (1 - \frac{\tau}{N - 1}) \cdot 100\% \tag{5}$$

Program	Faulty Versions	Blocks	Test Cases	Description
print_tokens	7	110	4 130	lexical analyzer
print_tokens2	10	105	4 115	lexical analyzer
replace	32	124	5 542	pattern recognition
schedule	9	53	2 650	priority scheduler
schedule2	10	60	2 710	priority scheduler
tcas	41	20	1 608	altitude separation
tot_info	23	44	1 052	information measure

Table 1. Set of programs used in the experiments

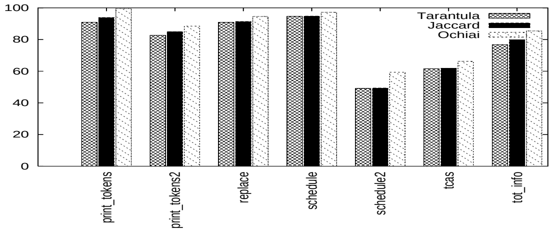

Figure 3. Diagnostic accuracy q_d

4 Similarity Coefficient Impact

At the end of Section 2.3 we reduced the problem of spectrum-based fault localization to finding resemblances between binary vectors. The key element of this technique is the calculation of a similarity coefficient. Many different similarity coefficients are used in practice, and in this section we investigate the impact of the similarity coefficient on the diagnostic accuracy q_d.

For this purpose, we evaluate q_d on all faults in our benchmark set, using nine different similarity coefficients. We only report the results for the Jaccard coefficient of Eq. (1), the coefficient used in the Tarantula fault localization tool as defined in Eq. (2), and the Ochiai coefficient of Eq. (3). We experimentally identified the latter as giving the best results among all eight coefficients used in a data clustering study in molecular biology [7], which also included the Jaccard coefficient.

In addition to Eq. (2), the Tarantula tool uses a second coefficient, which amounts to the maximum of the two fractions in the denominator of Eq. (2). This second coefficient is interpreted as a *brightness* value for visualization purposes, but the experiments in [16] indicate that the above coefficient can be studied in isolation. For this reason, we have not taken the brightness coefficient into account.

Figure 3 shows the results of this experiment. It plots q_d, as defined by Eq. (5), for the three similarity coefficients mentioned above, averaged per program of the Siemens set. See [1] for more details on these experiments.

An important conclusion that we can draw from these results is that under the specific conditions of our experiment, the Ochiai coefficient gives a better diagnosis: it always performs at least as good as the other coefficients, with an average improvement of 5% over the second-best case, and improvements of up to 30% for individual faults. Factors that likely contribute to this effect are the following. First, for $a_{11}(j) > 0$ (the only relevant case: $a_{11}(j) = 0$ implies $s_j = 0$) the Tarantula coefficient can be written as $1/(1 + c\frac{a_{10}(j)}{a_{11}(j)})$, with c the constant $\frac{a_{11}(j)+a_{01}(j)}{a_{00}(j)+a_{10}(j)}$. This depends only on presence of a block in passed and failed runs, while the Ochiai coefficient also takes the absence in failed runs into account. Second, compared to Jaccard (Eq. 1), for the purpose of determining the ranking the denominator of the Ochiai coefficient contains an extra term $\frac{a_{01}(j)\cdot a_{10}(j)}{a_{11}(j)}$, which amplifies the differences in the column vectors of Figure 2. This can be seen by squaring Eq. 3, and dividing the numerator and denominator by $a_{11}(j)$, which does not change the ranking.

5 Observation Quality Impact

Before reaching a definitive decision to prefer one similarity coefficient over another, as suggested by the results in Section 4, we want to verify that the effect of this decision is independent of specific conditions of our experiments. Because of its relation to test coverage, and to the error detection mechanism used to characterize runs as passed or failed, an important condition in this respect is the quality of the error detection information used in the analysis.

In this section we define a measure of quality of the error observations, and show how it can be controlled as a parameter if the fault location is known, as is the case in our experimental setup. Thus, we verify the results of the previous section for varying observation quality values. Investigating the influence of this parameter will also help us to assess the potential gain of more powerful error detection mechanisms and better test coverage on diagnostic accuracy.

5.1 A Measure of Observation Quality

Correctly locating the fault is trivial if the column for the faulty part in the matrix of Figure 2 resembles the error vector exactly. This would mean that an error is detected if, and only if the faulty part is active in a run. In that case, any coefficient is bound to deliver a highly accurate diagnosis. However, spectrum-based fault localization suffers from the following phenomena.

- Most faults lead to an error only under specific input conditions. For example, if a conditional statement contains the faulty condition $v < c$, with v a variable and c a constant, while the correct condition would be $v \leq c$, no error occurs if the conditional statement is executed, unless $v = c$.

- Similarly, as we have already seen in Section 2.1, errors need not propagate all the way to failures [18, 21], and may thus go undetected. This effect can partially be remedied by applying more powerful error detection mechanisms, but for any realistic software system and practical error detection mechanism there will likely exist errors that go undetected.

As a result of both phenomena, the set of runs in which an error is detected will only be a subset of the set of runs in which the fault is activated[2]. We propose to use the ratio of the size of these two sets as a measure of observation quality for a diagnosis problem. Using the notation of Section 2.3, we define

$$q_e = \frac{a_{11}(d)}{a_{11}(d) + a_{10}(d)} \cdot 100\%. \qquad (6)$$

This value can be interpreted as the unambiguity of the passed/failed data in relation to the fault being exercised, which may be loosely referred to as "error detection quality," hence the symbol q_e.

A problem with the q_e measure is that no information on undetected errors is available: $a_{10}(d)$ counts both the undetected errors, and the number of times the fault location was activated without introducing an error. This can be summarized as follows, where X, E, and D denote activation of the fault location, the occurrence of an error, and detection of an error, respectively:

X	E	D	
0	0	0	$a_{00}(d)$
1	0	0	$a_{10}(d)$
1	1	0	
1	1	1	$a_{11}(d)$

Even though the ratio of the two contributions to $a_{10}(d)$ is unknown, it can still be influenced in our experimental setup. We will now describe our procedure for doing so.

5.2 Varying q_e

Subject to various factors such as the nature of the fault, the similarity coefficient used in the diagnosis, the design of the test data, but also the compiler and the operating system, each faulty version of a program in our benchmark set has an inherent value for q_e, which can be evaluated by collecting spectra and error detection information for all available test cases, and performing the diagnosis of Section 2.3. In our environment, this inherent value for q_e ranges from 1.4% for `schedule2` to 20.3% for `tot_info`.

We can construct a different value for q_e by excluding runs that contribute either to $a_{11}(d)$ or to $a_{10}(d)$ as follows.

- Excluding a run that activates the fault location, but for which no error has been detected lowers $a_{10}(d)$, and will *increase* q_e.

- Excluding a run that activates the fault location and for which an error has been detected lowers $a_{11}(d)$, and will *decrease* q_e.

Excluding runs to achieve a certain value of q_e raises the question of which particular selection of runs to use. For this purpose we randomly sample passed or failed runs from the set of available runs to control q_e within a 99% confidence interval. We verified that the variance in the values measured for q_d is negligible.

Note that for decreasing q_e, i.e., obscuring the fault location, we have another option: setting failed runs to 'passed.' In our experiments we have tried both options, but the results were essentially the same. The results reported below are generated by excluding failed runs. Conversely, setting passed runs that exercise the fault location to 'failed' is not a good alternative for increasing q_e: this may obstruct the diagnosis as we cannot be certain that an error occurs for a particular data input. Moreover, it may allocate blame to parts of the program that are not related to the fault. Thus, excluding runs is always to be preferred as this does not compromise observation consistency. This way, we were able to vary q_e from 1% to 100% for all programs.

5.3 Similarity Coefficients Revisited

Using the technique for varying q_e introduced above we revisit the comparative study of similarity coefficients

[2] In our experimental setup, we do not consider effects that carry over from one run to another, so conversely, if an error is detected, the fault is always active.

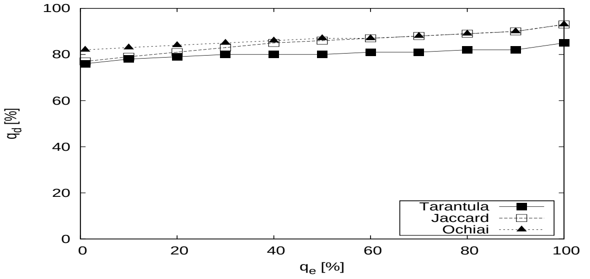

Figure 4. Observation quality impact

in Section 4. Figure 4 shows q_d for the three similarity coefficients, and values of q_e ranging from 1% to 100%. In this case, instead of averaging per program in the Siemens set, as we did in Figure 3, we arithmetically averaged q_d over all 120 faulty program versions to summarize the results (this is valid because q_d is already normalized with respect to program size). As in Figure 3, the graphs for the individual programs are similar, only having different offsets.

These results confirm what was suggested by the experiment in Section 4. The Ochiai similarity coefficient leads to a better diagnosis than the other eight, including the Jaccard coefficient and the coefficient of the Tarantula tool. Compared to the Jaccard coefficient the improvement is greatest for lower observation quality. As q_e increases, the performance of the Jaccard coefficient approaches that of the Ochiai coefficient. The improvement of the Ochiai coefficient over the Tarantula coefficient appears to be structural.

Another observation that can be made from Figure 4 is that all three coefficients provide a useful diagnosis (q_d around 80%) already for low q_e values (a q_e of 1% implies that only around 1% of the runs that exercised the faulty block actually resulted in a failed run). The accuracy of the diagnosis increases as the quality of the error detection information improves, but the effect is not as strong as we expected. This suggests that more powerful error detection mechanisms, or test sets that cover more input conditions will have limited gain. In the next section we investigate a possible explanation, namely that not only the quality of observations, but also their quantity determines the accuracy of the diagnosis.

6 Observation Quantity Impact

To investigate the influence of the number of runs on the accuracy of spectrum-based fault localization, we evaluated q_d while varying the numbers of passed (N_P) and failed runs (N_F) that are involved in the diagnosis, across the benchmark set. Since all interesting effects appear to occur for small numbers of runs, we have focused on the range of 1..100 passed and failed runs. Although the number of available runs in the Siemens set ranges from 1052 (`tot_info`) to 5542 (`replace`), the number of runs that fail is comparatively small, ranging from a single run for `tcas` version 8, to 518 for `print_tokens` version 2. For this reason, even in the range 1..100, some selections of failed runs are not possible for some of the faulty versions.

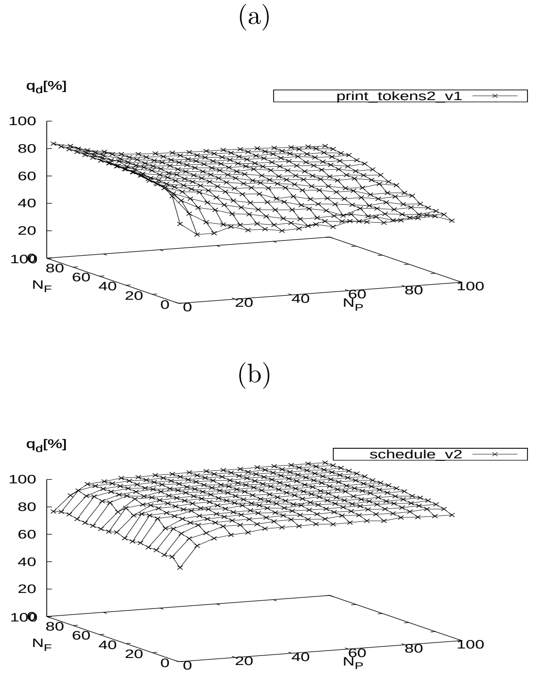

Figure 5. Observation quantity impact

Figure 5 shows two representative examples of such evaluations, where we plot q_d according to the Ochiai coefficient for N_P and N_F varying from 1 to 100. For each entry in these graphs, we averaged q_d over 50 randomly selected combinations of N_P passed runs and N_F failed runs, where we verified that the variance in the measured values of q_d is negligible. Apart from the apparent monotonic increase of q_d with N_F, we observe that for version 1 of `print_tokens2`, q_d decreases when more passed runs are added (Figure 5 (a)), while q_d increases for version 2 of `schedule` (Figure 5 (b)).

Given a set of faulty program versions that all allow failed runs to be selected up to a given value for N_F, we can average the measured values for q_d again over these versions. This summarizes several graphs of the kind shown in Figure 5. This way, in Figure 6 we plot the average q_d using the Ochiai coefficient for $1 \leq N_F \leq 30$ and $1 \leq N_P \leq 100$, projected on the $N_F \times q_d$ plane. With this limited range for N_F we can still use 80 of the

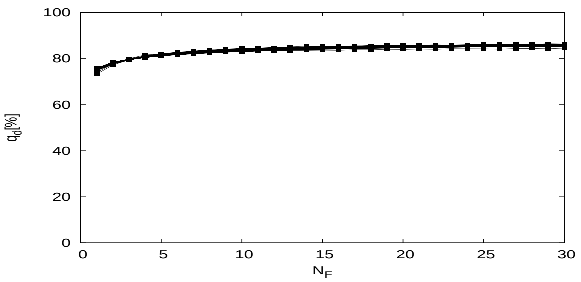

Figure 6. Impact of N_F on q_d, on average

118 versions in the benchmark set, whereas for $N_F \leq 100$, we can only use 35. We verified that for $N_F \leq 15$, for which we can use 95 versions, the results are essentially the same.

A first conclusion that we draw from Figure 6 is that overall, adding failed runs improves the accuracy of the diagnosis. However, the benefit of having more than 6 runs is marginal on average. In addition, because the measurements for varying N_P show little scattering in the projection, we can conclude that on average, N_P has little structural influence.

Inspecting the results for the individual program versions confirms our observation that adding failed runs consistently improves the diagnosis. However, although the effect does not show on average, N_P can have a significant effect on q_d for individual runs. As shown in Figure 5, this effect can be negative or positive. This shows more clearly in Figures 7 and 8, which contain cross sections of the graphs in Figure 5 at $N_F = 6$. To factor out any influence of N_F, we have created similar cross sections at the maximum number of failed runs. Across the entire benchmark set, we found that the effect of adding more passed runs stabilizes around $N_P = 20$.

Returning to the influence of the similarity coefficient once more, Figures 7 and 8 further indicate that the superior performance of the Ochiai coefficient is consistent also for varying numbers of runs. We have not plotted q_d for the other coefficients in Figure 5, but we verified this observation for all program versions, with N_P and N_F varying from 1 to 100.

From our experiments on the impact of the number of runs we can draw the following conclusions. First, including more failed runs is safe because the accuracy of the diagnosis either improves or remains the same. This is observed due to the fact that failed runs add evidence about the block that is causing the program to fail, and hence causing it to move up in the ranking. Our results show that the optimum value for N_F is roughly 6. To what extent this result depends on characteristics of the fault or program is subject to further investigation. Second, while stabilizing around $N_P = 20$, the effect of including more passed runs is unpredictable, and may actually decrease q_d. Actually, q_d decreases only if the faulty block is touched often in passed runs, as spectrum-based fault localization works under de assumption that if a block is touched often in passed runs, it should be exonerated. Besides, a large number of runs can apparently compensate weak error detection quality: even for small q_e, a large amount of runs provides sufficient information for good diagnostic accuracy, as shown in Figure 4. Lastly, the number of runs has no influence on the superiority of the Ochiai coefficient.

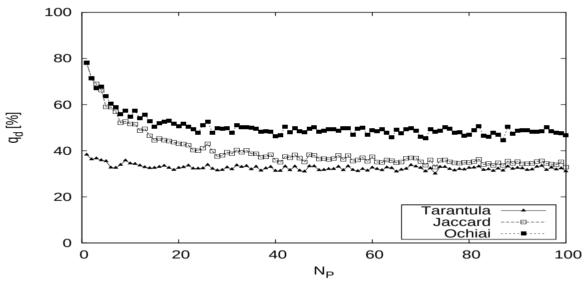

Figure 7. Impact of N_P on q_d for `print_tokens2` v1, and $N_F = 6$

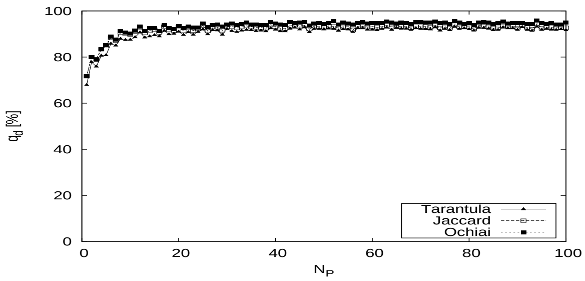

Figure 8. Impact of N_P on q_d for `schedule` v2, and $N_F = 6$

7 Related Work

Program spectra themselves were introduced in [20], where hit spectra of intra-procedural paths are analyzed to diagnose year 2000 problems. The distinction between count spectra and hit spectra is introduced in [13], where several kinds of program spectra are evaluated in the context of regression testing.

In the introduction we already mentioned three practical diagnosis/debugging tools [6, 8, 17] that are essentially based on spectrum-based fault localization. Pinpoint [6] is a framework for root cause analysis on the J2EE platform and is targeted at large, dynamic Internet services, such as web-mail services and search engines. The error detection is based on information coming from the J2EE framework, such as caught exceptions. The Tarantula tool [17] has been developed for the C language, and works with statement hit spectra. AMPLE [8] is an Eclipse plug-in for identifying faulty classes in Java software. However, although we have recognized that it uses hit spectra of method call sequences, we didn't include its weight coefficient in our experiments because the calculated values are only used to collect evidence about classes, not to identify suspicious method call sequences.

Diagnosis techniques can be classified as white box or black box, depending on the amount of knowledge that is required about the system's internal component structure and behavior. An example of a white box technique is model-based diagnosis (see, e.g., [9]), where a diagnosis is obtained by logical inference from a formal model of the system, combined with a set of run-time observations. White box approaches to software diagnosis exist (see, e.g., [22]), but software modeling is extremely complex, so most software diagnosis techniques are black box. Since the technique studied in this paper requires practically no information about the system being diagnosed, it can be classified as a black box technique.

Examples of other black box techniques are Nearest Neighbor [19], dynamic program slicing [3], and Delta Debugging [23]. The Nearest Neighbor technique first selects a single failed run, and computes the passed run that has the most similar code coverage. Then it creates the set of all statements that are executed in the failed run but not in the passed run. Dynamic program slicing narrows down the search space to the set of statements that influence a value at a program location where the failure occurs (e.g., an output variable). Delta Debugging compares the program states of a failing and a passing run, and actively searches for failure-inducing circumstances in the differences between these states. In [11] Delta Debugging is combined with dynamic slicing in 4 steps: (1) Delta Debugging is used to identify the minimal failure-inducing input; step (2) computes the forward dynamic slice of the input variables obtained in step 1; (3) the backward dynamic slice for the failed run is computed; (4) finally it returns the intersection of the slices given by the previous two steps. This set of statements is likely to contain the faulty code.

To our knowledge, none of the above approaches have evaluated diagnostic accuracy or studied the performance of similarity coefficients in the context of varying observation quality and quantity.

8 Conclusions and Future Work

Reducing fault localization effort greatly improves the test-diagnose-repair cycle. In this paper, we have investigated the influence of different parameters on the accuracy of the diagnosis delivered by spectrum-based fault localization. Our starting point was a previous study on the influence of the similarity coefficient, which indicated that the Ochiai coefficient, known from the biology domain, can give a better diagnosis than eight other coefficients, including those used by the Pinpoint [6] and Tarantula [16] tools.

By varying the quality and quantity of the observations on which the fault localization is based, we have established this result in a much wider context. We conclude that the superior performance of the Ochiai coefficient in diagnosing single-site faults in the Siemens set is consistent, and does not depend on the quality or quantity of observations. We expect that this result is relevant for the Tarantula tool, whose analysis is essentially the same as ours.

In addition, we found that even for the lowest quality of observation that we applied ($q_e = 1\%$, corresponding to a highly ambiguous error detection), the accuracy of the diagnosis is already quite useful: around 80% for all the programs in the Siemens set, which means that on average, only 20% of the code remains to be investigated to locate the fault. Furthermore, we conclude that while accumulating more failed runs only improves the accuracy of the diagnosis, the effect of including more passed runs is unpredictable. With respect to failed runs we observe that only a few (around 6) are sufficient to reach near-optimal diagnostic performance. Adding passed runs, however, can both improve or degrade diagnostic accuracy. In either case, including more than around 20 passed runs has little effect on the accuracy. The fact that a few observations can already provide a near-optimal diagnosis enables the application of spectrum-based fault localization methods within continuous (embedded) processing, where only limited observation horizons can be maintained.

In addition to our benchmark studies on the Siemens set, we have also evaluated spectrum-based fault localization on a large-scale industrial code (embedded software in consumer electronics, [24]). Based on the success of these exploratory experiments, new experiments are being defined that are much closer to the actual development process of our industrial partner in the TRADER project [10].

In future work, we plan to study the influence of the granularity (statement, function level) of program spectra on the diagnostic accuracy of spectrum-based fault localization. Furthermore, we intend to investigate the accuracy improvement of integrating static and dynamic program slicing (see, e.g., [13]) within our technique. Finally, our study was conducted using single fault programs, and further investigation is required to be able to generalize our findings to multiple fault programs.

9 Acknowledgments

We gratefully acknowledge the fruitful discussions with our TRADER project partners from Philips, NXP Semiconductors, Design Technology Institute, Embedded Systems Institute, IMEC, Leiden University, and Twente University.

References

[1] R. Abreu, P. Zoeteweij, and A. J. C. van Gemund. An evaluation of similarity coefficients for software fault localization. In *Proc. PRDC'06*, pp. 39 – 46, Riverside, CA, USA, December 2006. IEEE Computer Society.

[2] R. Abreu, P. Zoeteweij, and A. J. C. van Gemund. Program spectra analysis in embedded systems: A case study. In *12th Ann. Conf. of the Advanced School for Computing and Imaging*. ASCI, June 2006.

[3] H. Agrawal, R. A. DeMillo, and E. H. Spafford. Debugging with dynamic slicing and backtracking. *Software - Practice and Experience*, 23(6):589–616, 1993.

[4] L. Augusteijn. Front: a front-end generator for Lex, Yacc and C, release 1.0. http://front.sourceforge.net/, 2002.

[5] 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.

[6] M. Y. Chen, E. Kiciman, E. Fratkin, A. Fox, and E. Brewer. Pinpoint: Problem determination in large, dynamic internet services. In *Proc. DSN'02*, pp. 595–604, Washington, DC, USA, 2002. IEEE Computer Society.

[7] A. da Silva Meyer, A. A. Franco Farcia, and A. Pereira de Souza. Comparison of similarity coefficients used for cluster analysis with dominant markers in maize (*Zea mays* L). *Genetics and Molecular Biology*, 27(1):83–91, 2004.

[8] V. Dallmeier, C. Lindig, and A. Zeller. Lightweight defect localization for Java. In *Proc. ECOOP'05*, volume 3568 of *LNCS*, pp. 528–550, Glasgow, UK, 2005. Springer-Verlag.

[9] J. de Kleer and B. C. Williams. Diagnosing multiple faults. *Artif. Intell.*, 32(1):97–130, 1987.

[10] Embedded Systems Institute. Trader project website. http://www.esi.nl/trader/.

[11] N. Gupta, H. He, X. Zhang, and R. Gupta. Locating faulty code using failure-inducing chops. In *Proc. of the 20th IEEE/ACM international Conference on Automated software engineering*, pp. 263–272, Long Beach, CA, USA, 2005. ACM Press.

[12] B. Hailpern and P. Santhanam. Software debugging, testing, and verification. *IBM Systems Journal*, 41(1):4–12, 2002.

[13] M. J. Harrold, G. Rothermel, R. Wu, and L. Yi. An empirical investigation of program spectra. In *Proc. of the SIGPLAN/SIGSOFT Workshop on Program Analysis For Software Tools and Engineering, Montreal, Canada, June 16, 1998*, pp. 83–90, 1998.

[14] M. Hutchins, H. Foster, T. Goradia, and T. Ostrand. Experiments of the effectiveness of dataflow- and controlflow-based test adequacy criteria. In *Proc. ICSE'94*, pp. 191–200, Sorrento, Italy, 1994. IEEE Computer Society.

[15] A. K. Jain and R. C. Dubes. *Algorithms for clustering data.* Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1988.

[16] J. A. Jones and M. J. Harrold. Empirical evaluation of the tarantula automatic fault-localization technique. In *Proc. ASE'05*, pp. 273–282, New York, NY, USA, 2005. ACM Press.

[17] J. A. Jones, M. J. Harrold, and J. Stasko. Visualization of test information to assist fault localization. In *Proc. ICSE'02, Orlando, Florida, USA, May 2002*, pp. 467–477. ACM Press.

[18] L. J. Morell. A theory of fault-based testing. *IEEE TSE*, 16(8):844–857, 1990.

[19] M. Renieris and S. P. Reiss. Fault localization with nearest neighbor queries. In *Proc. ASE'03*, Montreal, Canada, October 2003. IEEE Computer Society.

[20] 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 *Proc. ESEC'97*, volume 1301 of *LNCS*, pp. 432–449. Springer–Verlag, 1997.

[21] J. Voas. PIE: A dynamic failure based technique. *IEEE TSE*, 18(8):717–727, August 1992.

[22] F. Wotawa, M. Strumptner, and W. Mayer. Model-based debugging or how to diagnose programs automatically. In *Proc. IAE/AIE 2002*, volume 2358 of *LNCS*, pp. 746–757. Springer-Verlag, 2002.

[23] A. Zeller. Isolating cause-effect chains from computer programs. In *Proc. FSE'02, Charleston, South Carolina, November 2002.* ACM Press.

[24] P. Zoeteweij, R. Abreu, R. Golsteijn, and A. J. C. van Gemund. Diagnosis of embedded software using program spectra. In *Proc. ECBS'07*, pp. 213–218, Tucson, AZ, USA, March 2007. IEEE Computer Society.