An Evaluation of Similarity Coefficients for Software 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

Automated diagnosis of software faults can improve the efficiency of the debugging process, and is therefore an important technique for the development of dependable software. In this paper we study different similarity coefficients that are applied in the context of a program spectral approach to software fault localization (single programming mistakes). The coefficients studied are taken from the systems diagnosis / automated debugging tools Pinpoint, Tarantula, and AMPLE, and from the molecular biology domain (the Ochiai coefficient). We evaluate these coefficients on the Siemens Suite of benchmark faults, and assess their effectiveness in terms of the position of the actual fault in the probability ranking of fault candidates produced by the diagnosis technique. Our experiments indicate that the Ochiai coefficient consistently outperforms the coefficients currently used by the tools mentioned. In terms of the amount of code that needs to be inspected, this coefficient improves 5% on average over the next best technique, and up to 30% in specific cases.

Keywords: *Software reliability, automated debugging, software fault diagnosis, fault localization, program spectra.*

1 Introduction

Software reliability can generally be improved through extensive testing and debugging, but this is often in conflict with market conditions: software cannot be tested exhaustively, and of the bugs that are found, only those with the highest impact on the user-perceived reliability can be solved before the release. In this typical scenario, testing reveals more bugs than can be solved, and debugging is the bottleneck for improving reliability. Automated debugging techniques can help to reduce this bottleneck. These techniques give a diagnosis for errors that are detected during the execution of a program, which can help programmers to locate their root causes, and thus to reduce the effort spent on manual debugging.

Diagnosis techniques, which include automated debugging, can be classified as white box or black box, depending on the amount of knowledge that is required about the system under study. An example of a white box technique is model-based diagnosis (see, e.g., [6, 7]), where a diagnosis is obtained by logical inference from a formal model of the system, combined with a set of run-time observations. While white box approaches to software diagnosis exist (see, e.g., [14, 17, 18, 19]), software modeling is extremely complex. Hence, most software diagnosis techniques are black box. The subject of this paper is a specific automated debugging technique, namely software fault localization through the analysis of program spectra. Because this technique requires practically no information about the system being diagnosed, it can be classified as a black box diagnosis technique.

Program spectra can be seen as projections of traces of software activity. Typically, this projection is much more compact than a trace, which makes program spectra an attractive technique in resource-constrained environments, such as embedded systems. The technique studied in this paper is based on analyzing the differences between program spectra obtained for correct behavior of the software, and program spectra obtained for faulty behavior of the software. An essential part of this analysis is the computation of a measure of similarity between different vectors in the program spectra data and a vector that contains information

*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.

about the detected errors. Different application reports of the technique are now emerging in the literature, but all use different similarity measures.

In this paper we investigate the influence of the similarity measure on the quality of the diagnosis. To this end we apply the fault localization technique on a benchmark set of software faults known as the *Siemens Suite*. We obtain multiple diagnoses for every fault in the suite, each of them for a different similarity measure. These similarity measures are taken from existing diagnosis tools in the areas of recovery and automated debugging, and from the molecular biology domain. A diagnosis for a particular measure of similarity consists of a list of possible locations for the fault ranked in order of similarity. Our evaluation is based on the position of the (known) location of the fault in this ranking. In particular, the contributions of the paper are the following.

- We recognize that several existing tools are implementations of the same technique: fault diagnosis through the comparison of program spectra, which allows us to compare the techniques.
- We identify a measure of similarity between the program spectra that yields an average performance improvement of 5% under the specific conditions of our experiments, in terms of the amount of code that must be inspected. Improvements up to 30% are measured.

The remainder of the paper is organized as follows. In Section 2 we introduce some basic concepts and terminology, illustrate the fault localization technique, and review related work. In Section 3 we introduce the similarity coefficients that we evaluate and compare. In Section 4 we describe our experimental setup, which includes a description of the benchmark suite. The results of our experiments are discussed in Section 5. We conclude in Section 6.

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 defined in [2], 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.

```
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

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

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

Error detection is a prerequisite for diagnosis. We must know that something is wrong before we can try to locate the fault. Failures constitute a rudimentary form of error detection, but many errors remain latent and never lead to a

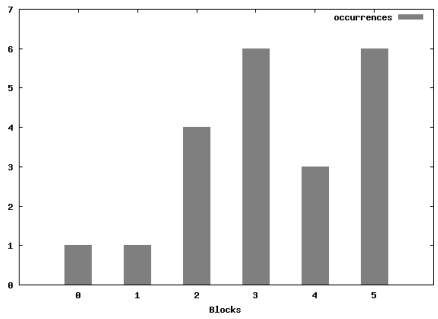

Figure 2. Block count spectrum

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 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 [16] 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. For example, *block count spectra* count how often every block 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 1, 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 1 is called from the following `main` function, to sort the sequence $\langle \frac{2}{1}, \frac{3}{1}, \frac{4}{1}, \frac{1}{1} \rangle$, which it happens to do correctly.

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

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

Running this program would result in the block count spectrum represented by the histogram in Figure 2. Blocks 0 and 1, the bodies of functions `main` and `RationalSort`, are both executed once. Blocks 2 and 3, the bodies of the two loops in `RationalSort` are executed four and six times, respectively. To sort the array in our example program we need to make three exchanges, and block 4, the `if` branch of the conditional statement, is executed three times. Block 5 has not been shown in Figure 1, but it represents the body of the `RationalGT` function. It is executed six times: once on every iteration of the inner loop.

If we are only interested in whether a block is executed or not, but not in the number of times it is executed, we can use binary flags instead of counters, and block count spectra revert to block *hit* spectra. Beside block count/hit spectra, many other forms of program spectra are used in practice. See [8] for an overview. In this paper we work with fine-grained *block hit spectra*.

2.3 Fault Diagnosis

Program spectra can be used for fault diagnosis by comparing spectra for runs in which an error has been detected (usually called *failed* runs), to spectra for runs in which no error has been detected (usually called *passed* runs), and analyzing the differences. Using block spectra, this may identify those blocks that are executed primarily in failed runs. These blocks 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 $S_1 = \langle \frac{1}{4}, \frac{1}{3}, \frac{1}{2}, \frac{1}{1} \rangle$ and $S_2 = \langle \frac{3}{1}, \frac{2}{2}, \frac{4}{3}, \frac{1}{4} \rangle$. The former sequence is already sorted, and the program will pass, but the latter sequence will result in a failure, which is a clear indication that an error has occurred. The block hit spectra for the two runs are as follows ('X' denotes a hit).

	block						
input	0	1	2	3	4	5	error
S_1	X	X	X	X		X	
S_2	X	X	X	X	X	X	X

The difference between the two block hit spectra (correctly) identifies block 4 as the most likely location of the fault: while all other blocks are executed in both runs, block 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 blocks 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. The block hit spectra of the various runs constitute a binary matrix, whose columns correspond to the blocks of the program. A column vector identifies in which runs a block has been activated. In some of the runs an error is detected. This information constitutes another column vector, the er-

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

ror vector. This vector can be seen as a hypothetical block of code that is responsible for all observed errors. Fault diagnosis essentially consists in identifying the block whose column vector resembles the error vector most. This last part, and particularly the measure, or coefficient, of similarity used to compare column vectors to the error vector, is the primary focus of this paper.

2.4 Related Work

The diagnosis approach described in Sections 2.2 and 2.3 has appeared in various guises in literature. Three systems are of particular interest, because the similarity coefficient that is used in the diagnosis is clearly described. They are Pinpoint, Tarantula, and AMPLE.

Pinpoint [3] is a framework for root cause analysis on the J2EE platform. It is developed in the context of the Recovery Oriented Computing project [15], and is targeted at large, dynamic Internet services, such as web-mail services and search engines. It combines the technique of the previous section with a specific form of error detection, based on information coming from the J2EE framework, such as caught exceptions, and errors visible to users, such as HTTP errors. This makes the approach self-contained in the sense that no external characterization of traces is needed.

The Tarantula system [12, 13] has been developed for the C language, and applies the technique of the previous section to statement hit spectra. Compared to block hit spectra, the higher resolution of statement hit spectra may give a more detailed diagnosis in presence of statements that alter the flow of control inside a block, namely `break`, `continue`, `return`, and `goto`. Tarantula comes with a graphical user interface, that interprets the calculated value for the similarity coefficient as a color index, used to visualize the suspiciousness of program statements. Tarantula relies on external error detection for the classification of runs as passed or failed: whereas Pinpoint uses information from the J2EE framework for this classification, this information is input data for Tarantula. In other words, Tarantula implements only the diagnosis, and has to be complemented by adding a method of error detection.

AMPLE (Analyzing Method Patterns to Locate Errors) [5] is a system for identifying faulty classes in object-oriented software. It collects hit spectra of *method call sequences*, which are subsequences of a given length that occur in a full trace of incoming or outgoing method calls, received or issued by individual objects of a class. Each call sequence is assigned a weight, which captures the extent to which its occurrence or absence correlates with the detection of an error, i.e., it is a combined measure of similarity and inverted similarity. These weights are averaged over all call sequences of a class, leading to a class weight. Classes with a high weight are most likely to contain the fault that causes the detected error. Although the calculation of the sequence weights in AMPLE can be explained as an application of the technique of Section 2.3, the diagnosis is at class level, and the calculated coefficients are used only to collect evidence about classes, not to identify suspicious method call sequences.

$$
\begin{array}{c} \\ M \text{ spectra} \end{array}
\begin{array}{c}
\begin{array}{cccc} \multicolumn{4}{c}{N \text{ blocks}} \end{array} \\
\begin{bmatrix} x_{11} & x_{12} & \dots & x_{1N} \\ x_{21} & x_{22} & \dots & x_{2N} \\ \vdots & \vdots & \ddots & \vdots \\ x_{M1} & x_{M2} & \dots & x_{MN} \end{bmatrix} \\
\begin{array}{cccc} s_1 & s_2 & \dots & s_N \end{array}
\end{array}
\quad
\begin{array}{c}
\text{error} \\ \text{detection} \\
\begin{bmatrix} e_1 \\ e_2 \\ \vdots \\ e_M \end{bmatrix}
\end{array}
$$

Figure 3. The ingredients of fault diagnosis

Concluding, we can observe that three existing tools for diagnosis and automated debugging rely on an analysis of program spectra. Program spectra themselves were introduced in [16], 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 [8], where several kinds of program spectra are evaluated in the context of regression testing. As we already mentioned in the introduction, in the context of computer programs, fault localization based on the analysis of program spectra is an automated debugging technique. An example of a different (black box) technique in that category is Delta Debugging [20], which compares the program states of a failing and a passing run, and actively searches for failure-inducing circumstances in the differences between these states.

3 Similarity Analysis

At the end of Section 2.3 we reduced the problem of fault diagnosis based on block hit spectra to finding resemblances between binary vectors: M different runs of the same software, for example, on different inputs, yield M block hit spectra. These spectra constitute the rows of an $M \times N$ binary matrix, whose columns correspond to the N blocks of the program. In some of the M runs an error is detected. Other runs complete without an error. This information yields another binary column vector, which can be thought of as to correspond to a hypothetical block of code that is responsible for all errors. For each block of the program, we evaluate the similarity s_j of its column vector with the column vector of error detection information. The block with the highest similarity is most likely to contain the fault (see Figure 3).

In the field of data clustering, similarity measures for binary, nominally scaled data, such as the vectors $x_{1j}, \dots, x_{Mj}$ and $e_1, \dots, e_M$ that we want to compare, are called *similarity coefficients*. Using the notation of [11], we

can express such similarity coefficients using four counters of pairs of values that can occur on any position i in these two vectors:

	values:	
counter	x_{ij}	e_i
a_{11}	1	1
a_{10}	1	0
a_{01}	0	1
a_{00}	0	0

For example, a_{11} is the number of positions i in which the j-th column vector and the error vector share an entry 1, i.e., the number of spectra in which block j was recorded to be executed, and for which an error has been detected. These four counters sum up to the number of spectra M.

In this paper we investigate the influence of the similarity coefficient on the quality of the diagnosis by applying a number of similarity coefficients known from the literature on the same data, and comparing the resulting diagnoses. We will evaluate the following similarity coefficients:

- Jaccard:
$$s_j = \frac{a_{11}}{a_{11} + a_{01} + a_{10}} \quad (1)$$
The Jaccard coefficient, which is well-known from the field of data clustering (see, e.g., [11]), is used in the Pinpoint framework [3].

- Tarantula:
$$s_j = \frac{\frac{a_{11}}{a_{11}+a_{01}}}{\frac{a_{11}}{a_{11}+a_{01}} + \frac{a_{10}}{a_{10}+a_{00}}} \quad (2)$$
This coefficient is used in the Tarantula system [13, 12].

- AMPLE:
$$s_j = \mid \frac{a_{11}}{a_{01} + a_{11}} - \frac{a_{10}}{a_{00} + a_{10}} \mid \quad (3)$$
This coefficient is used by the AMPLE tool. In [5] it is assumed that there is exactly one failing run, in which case the denominator $a_{01} + a_{11}$ equals 1.

- Ochiai:
$$s_j = \frac{a_{11}}{\sqrt{(a_{11} + a_{01}) * (a_{11} + a_{10})}} \quad (4)$$
This coefficient is used in [4] for computing genetic similarity in molecular biology.

In addition to the coefficient of Eq. (2), the Tarantula system 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 by its visualization system, but the experiments in [12] indicate that the above coefficient can be studied in isolation. For this reason, we have not taken the brightness coefficient into account.

The AMPLE coefficient is used here outside its context. It amounts to the relative *difference* between the number of occurrences of a block in passed and failed runs, and hence also takes absence of a block in failing runs into account. This probably has little use without accumulating the calculated values to a coarser-grained unit of diagnosis (cf., accumulating call sequence weights to class weights, see Section 2.4), and therefore one should not project our results for this coefficient to the AMPLE tool.

Several other similarity coefficients are used in data clustering (see [4] for a study in the context of molecular biology). Although all the coefficients presented in [4] were used in our experiments, in this paper we have only included the Ochiai coefficient, which gave the best results.

4 Experimental Setup

In this section we introduce the benchmark used in the experiments and the technical details related to data acquisition. Finally we define quality of diagnosis as evaluation metric.

4.1 Benchmark Set

Evaluating different similarity coefficient techniques requires us to thoroughly test them. For this purpose we used a set of test programs known as the *Siemens suite* [9]. The Siemens suite 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. Those inputs were created with the intention to test the full coverage of the programs. Table 1 provides more information about the programs in the package (for more detailed information refer to [9]). Although the Siemens suite was not assembled with the purpose of testing fault diagnosis techniques, it is typically used by the research community as the set of programs to test their techniques.

In our experiments we were not able to use all the programs provided by the Siemens suite. Because we conduct our experiments using block hit spectra, we can not use programs which contain data-dependent faults, i.e., faults that do not influence control flow. Versions 4 and 6 of *print_tokens*, version 38 of *tcas*, and version 10 of *tot_info* contain errors that are considered to be data dependent 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*,

Program	Faulty Versions	Blocks	Test Cases	Description
print_tokens	7	110	4056	lexical analyzer
print_tokens2	10	105	4071	lexical analyzer
replace	32	124	5542	pattern recognition
schedule	9	53	2650	priority scheduler
schedule2	10	60	2680	priority scheduler
tcas	41	20	1578	altitude separation
tot_info	23	44	1054	information measure

Table 1. Description of the Siemens Suite

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 14 versions out of 132 versions provided by the suite, using 118 versions in our experiments.

4.2 Data Acquisition

Collecting Spectra For obtaining block hit spectra we instrumented the source code of every single program in the Siemens suite. A function call was automatically inserted in the beginning of every block of code to log its execution. To automatically instrument the programs, we use a tool called *Front* [1]. Moreover, the programs were compiled on a Linux based environment with gcc-3.2.

Error Detection As for each program the Siemens suite 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. As we explained in Section 2.1, failure detection is a rudimentary form of error detection where a faulty program may well go undetected. Using the notation from Section 3, we define error detection accuracy for a given faulty block as

$$q_e = \frac{a_{11}}{a_{11} + a_{10}} \tag{5}$$

With our strategy to detect failing and passing runs, we cannot expect good error detection accuracy. On average per program, in the Siemens set the error accuracy ranges from 1.2% (*schedule2*) to 21.1% (*tot_info*). This implies that measured diagnostic quality will be limited due to this low error detection accuracy.

4.3 Evaluation Metric

For the purpose of this discussion, we define quality of the diagnosis as the position that the faulty block has in the ranking. The notion behind this measure is how many blocks we still need to inspect until we identify the faulty block. If there are two blocks that rank with the same coefficient, we use the worst ranking position for both of them.

More precisely, 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 diagnostic quality is given by

$$q_d = |\{j | s_j \geq s_d\}| \tag{6}$$

5 Experimental Results

This section presents results obtained by applying the coefficients of similarity described in Section 3 to the data collected by the execution of the benchmark programs. Under the assumption that a high similarity to the error vector of Section 3 indicates a high probability of a block containing a fault, the calculated coefficients of similarity result in a ranking of blocks, sorted by likelihood of causing the error. The intention of this experiment is to find which of the coefficient of similarity leads to the better diagnosis.

In Table 2 we show the average ranking position of the actual fault for the different similarity functions (Equations (1) through (4)), per program. For each program we run all the versions (faults) on all test cases, and rank the blocks according to Equations (1) through (4). Per equation we determine the position q_d of the faulty block in the ranking, and average this number over all versions of the program. We do not average over the seven programs because they have largely varying numbers of blocks.

One significant observation from this table is that the Ochiai coefficient (Eq. (4)) consistently outperforms the other techniques in terms of diagnostic quality. The Jaccard coefficient and the coefficient used by Tarantula sometimes happen to have the same quality of diagnosis. However, in all except one of the situations where they differ we observe that Jaccard is better than Tarantula's coefficient. AMPLE's coefficient produces the worst quality of diagnosis except for two programs where the quality was better than Jaccard and the coefficient used by Tarantula. In one of these programs, the AMPLE coefficient yields similar quality compared to the Ochiai coefficient. As noted in Section 2.4, we are using the AMPLE coefficient outside its original context, and hence, this does not imply that the AMPLE system performs worse than the Tarantula system.

	Tarantula	Jaccard	Ochiai	AMPLE
print_tokens	10.5	7.3	1.0	2.0
print_tokens2	20.0	17.5	13.9	16.4
replace	12.2	11.6	7.6	12.7
schedule	3.0	2.9	1.6	11.3
schedule2	31.3	31.1	25.1	34.7
tcas	8.8	8.8	7.9	9.8
tot_info	11.0	9.6	7.1	13.2

Table 2. Average fault ranking position

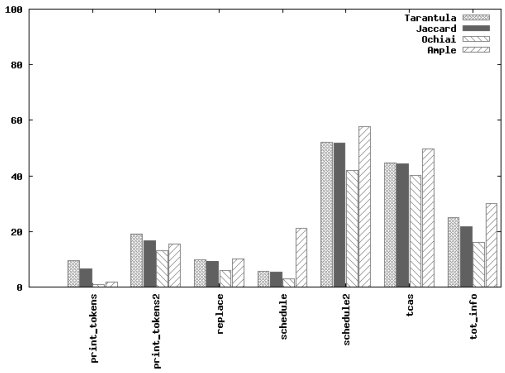

Figure 4. Percentage of blocks to be inspected

As said, automated fault diagnosis aims to decrease fault localization effort. In this respect, a good measure of quality is the *percentage* of blocks a programmer has to inspect until he/she finds the bug, assuming that the programmer would inspect the code according to the ranking created by the diagnosis technique. This percentage is defined by the average rank of the fault divided by the total number of blocks, and is shown in Figure 4. From this figure it is immediately clear that, under the specific conditions of our experiments, the Ochiai coefficient is superior. Using this coefficient of similarity, in the worst case of this experiment, the software developer is still 'obliged' to inspect 40% of the code to find the fault. In the best case, only 1% needs to be inspected. The Ochiai coefficient presents improvements ranging from 2.4% to 10% on average per program over the Jaccard coefficient (second-best technique). Per faulty version, improvements up to 30% were measured. Overall, this coefficient decreases the percentage of blocks of code to be inspected by 5%.

Figure 4 also shows that all of the coefficients work poorly for some programs, for instance *schedule2* and *tcas*. The poor ranking is mainly due to dependent blocks, i.e., blocks that are always executed when the faulty block is also executed. Hence, these blocks will rank in the same position, worsening the quality of diagnosis q_d.

6 Conclusions and Future Work

In this paper we studied the influence of different similarity coefficients for fault diagnosis using program hit spectra at the block level. We conclude that using this type of program spectra the Ochiai coefficient consistently outperforms those used in Pinpoint (Jaccard), Tarantula, and AMPLE. Apart from the performance improvement (ranging from 2.6% to 10%), the fact that this similarity coefficient is outside the software fault diagnosis domain is somewhat unexpected, and suggests that further improvements may even exist.

Specifically for the Tarantula system, our experiments seem to suggest that its diagnostic capabilities can be improved by switching to the Ochiai coefficient. Tarantula uses statement hit spectra, which give more detailed information than our block hit spectra, but this will only lead to a different diagnosis in in case of the C constructs mentioned in Section 2.4. Apart from this, our approach is the same as that of Tarantula, and our results should be applicable.

Furthermore, the experiments also revealed some versions yielding less effective rankings. The possible causes for poor rankings are, mainly, when the faulty block is always exercised in passed and failed runs (for instance, the *main* function), deleted code, and dependent blocks as explained in the previous section.

Future work will address the following issues. Unlike the rudimentary error detection mechanism used in this research we will investigate the influence of q_e on q_d in order to obtain a more general picture on the performance of the various similarity coefficients. Furthermore, we intend to extend our study to multiple faults. Finally, we will involve a much larger code base in our study. In a recent study of a mega-LOC industrial code (Philips TV software, in the context of the TRADER project [10]), we found that all techniques behaved extremely well (sometimes with $q_d = 1$ out of 65K blocks), which is partly due to an ideal error detection mechanism. This suggests that the Siemens suite may be not entirely representative for real-world (embedded) software.

Acknowledgments

We gratefully acknowledge the feedback from the discussions with our TRADER project partners from Philips Research, Philips Semiconductors, Philips TASS, Philips Consumer Electronics, Design Technology Institute, Embedded Systems Institute, IMEC, Leiden University and Twente University.

References

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

[2] 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.

[3] M. Y. Chen, E. Kiciman, E. Fratkin, A. Fox, and E. Brewer. Pinpoint: Problem determination in large, dynamic internet services. In *Proceedings of the 2002 International Conference on Dependable Systems and Networks*, pages 595–604, Washington, DC, USA, 2002. IEEE Computer Society.

[4] 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.

[5] V. Dallmeier, C. Lindig, and A. Zeller. Lightweight defect localization for Java. In A. P. Black, editor, *ECOOP 2005 : 19th European Conference, Glasgow, UK, July 25–29, 2005. Proceedings*, volume 3568 of *LNCS*, pages 528–550. Springer-Verlag, 2005.

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

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

[8] M. J. Harrold, G. Rothermel, R. Wu, and L. Yi. An empirical investigation of program spectra. In *Proceedings of the SIGPLAN/SIGSOFT Workshop on Program Analysis For Software Tools and Engineering, PASTE '98, Montreal, Canada, June 16, 1998*, pages 83–90, 1998.

[9] M. Hutchins, H. Foster, T. Goradia, and T. Ostrand. Experiments of the effectiveness of dataflow- and controlflow-based test adequacy criteria. In *Proceedings of the 16th international conference on Software engineering*, pages 191–200, Sorrento, Italy, 1994. IEEE Computer Society Press.

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

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

[12] J. A. Jones and M. J. Harrold. Empirical evaluation of the tarantula automatic fault-localization technique. In *Proceedings of the 20th IEEE/ACM international Conference on Automated software engineering*, pages 273–282, New York, NY, USA, 2005. ACM Press.

[13] J. A. Jones, M. J. Harrold, and J. Stasko. Visualization of test information to assist fault localization. In *Proceedings of the 24th International Conference on Software Engineering, Orlando, Florida, USA, May 2002*, pages 467–477. ACM Press, 2002.

[14] W. Mayer and M. Stumptner. Approximate modeling for debugging of program loops. In *Proceedings of the Fifteenth International Workshop on Principles of Diagnosis*, Carcassonne, June 2004.

[15] D. Patterson, A. Brown, P. Broadwell, G. Candea, M. Chen, J. Cutler, P. Enriquez, A. Fox, E. Kiciman, M. Merzbacher, D. Oppenheimer, N. Sastry, W. Tetzlaff, J. Traupman, and N. Treuhaft. Recovery Oriented Computing (ROC): Motivation, definition, techniques, and case studies. Technical Report UCB/CSD-02-1175, U.C. Berkeley, March 2002.

[16] T. Reps, T. Ball, M. Das, and J. Larus. The use of program profiling for software maintenance with applications to the year 2000 problem. In M. Jazayeri and H. Schauer, editors, *Proceedings of the Sixth European Software Engineering Conference (ESEC/FSE 97)*, volume 1301 of *LNCS*, pages 432–449. Springer–Verlag, 1997.

[17] M. Stumptner. Using design information to identify structural software faults. In *Proceedings of the 14th Australian Joint Conference on Artificial Intelligence*, volume 2256 of *LNCS*, pages 473–486, London, UK, 2001. Springer-Verlag.

[18] F. Wotawa. On the relationship between model-based debugging and program slicing. *Artificial Intelligence*, 135(1-2):125–143, 2002.

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

[20] A. Zeller. Isolating cause-effect chains from computer programs. In *Proceedings of the 10th International Symposium on the Foundations of Software Engineering (FSE-10), Charleston, South Carolina, November 2002*. ACM Press, 2002.