2012 IEEE Sixth International Conference on Software Security and Reliability

Software Fault Localization Using DStar (D^*)

W. Eric Wong[1], Vidroha Debroy[2], Yihao Li,[1] and Ruizhi Gao[1]
[1]Department of Computer Science, University of Texas at Dallas, USA
[2]Microsoft Corporation, Redmond, Washington, USA

Abstract—Effective debugging is crucial to producing dependable software. Manual debugging is becoming prohibitively expensive, especially due to the growing size and complexity of programs. Given that fault localization is one of the most expensive activities in program debugging, there has been a great demand for fault localization techniques that can help guide programmers to the locations of faults. In this paper a technique named DStar (D^*), which has its origins rooted in similarity coefficient-based analysis, is proposed; which can identify suspicious locations for fault localization automatically without requiring any prior information on program structure or semantics. D^* is evaluated across 21 programs and is compared to 16 different fault localization techniques. Both single-fault and multi-fault programs are used. Results indicate that D^* is more effective at locating faults than all the other techniques it is compared to.

Index Terms— Testing, debugging, software fault localization

1. Introduction

It is a well recognized fact that manual debugging is not just time-consuming, tedious and expensive, but is also quite error-prone [11,17,29]. Thus, approaches that can help automate the debugging process, thereby making software more dependable and maintainable, are currently in great demand. Among the various debugging activities, software[1] fault localization (hereafter, referred to simply as 'fault localization'), which deals with finding the locations of faults, has been identified as one of the most expensive [24]. Even in an educational setting, based on interactions with students, it has been reported that finding faults in program code is an extremely difficult task [10]. But fault localization is also exceedingly important as the faster the source of an error can be found, the faster its cause can be addressed; and this reduces the amount of downtime a system may incur, thereby improving availability [13]. Such realizations have paved the way for the proposal and development of a number of different fault localization techniques over the recent years, each of which tries to identify the locations of faults in one way or the other [1,2,7,15,16,17,20,26,33].

The general idea is for a fault localization technique to create a ranking of most probable faulty components (e.g., program statements or predicates) such that these components may then be examined by programmers in order of their suspiciousness (likelihood of containing faults) until a fault is found. A good technique should rank a faulty component towards the top (if not at the very top) of its ranking such that a fault is discovered early on in the examination of the ranking. In fact if the faulty component is at the very top of the ranking (i.e., it is the first component that is examined) then the programmer has been led to the location of a fault purely automatically, without having to intervene thus far.

In this paper we propose a new fault localization technique which has its origins rooted in binary similarity coefficient-based analysis. We propose the use of a modified form of the *Kulczynski* coefficient [6,25] in the context of fault localization. We acknowledge that often the same coefficient is referred to by different names in different contexts (e.g., the *Ochiai*, *Cosine* and *Otsuka* coefficients are all algebraically one and the same [6,25]) and thus the modified version of the coefficient may already be in existence. However, to the best of our knowledge such a coefficient has not been proposed before (certainly not in the context of fault localization), and hence we name the new coefficient – 'DStar' (hereafter, abbreviated simply to D^*), where the value of * may vary (see Section 4 for more details).

We systematically evaluate D^* across 6 different sets of subject programs (*Siemens suite*, *Unix suite*, *grep*, *gzip*, *make* and *Ant*) each consisting of multiple different faulty versions. First, the proposed modification to the Kulczynski coefficient (i.e., modifying it to D^*) is justified (in the context of fault localization) by the fact that D^* leads to *better* (more effective) fault localization results than the use of the standard Kulczynski coefficient. We formally define what we mean by 'better' in Section 3.3. Second, the use of D^* as a fault localization technique is justified by the fact that it performs better than 16 other techniques (each described in detail subsequently in this paper) not just limited to similarity coefficient-based techniques.

The contributions of this paper can be summarized as:

1. A fault localization technique named D^* is proposed based on a modification of the Kulczynski coefficient.
2. Effectiveness of D^* is evaluated across 21 programs, and compared to 16 different fault localization techniques. It is demonstrated that D^* is better (or more effective) at fault localization than the other techniques.

The rest of the paper is organized as follows. Section 2 overviews similarity coefficient-based fault localization, and presents D^* with an illustrative example. Section 3 discusses our case studies, reporting details on the experimental design, data collection, etc., before moving on to the results. Section 4 shows how the effectiveness of D* improves and levels off as the value of the * increases. Then, D^* is further compared to other fault localization techniques in Section 5. Section 6 addresses programs with multiple faults, followed by Section 7 which describes work related to the subject presented in this paper. Finally, we provide our conclusions and plans for future work in Section 8.

[1] In this paper, we use the terms "program" and "software" interchangeably. We also use "bugs" and "faults" interchangeably.

978-0-7695-4742-8/12 $26.00 © 2012 IEEE
DOI 10.1109/SERE.2012.12

2. The Proposed Fault Localization Technique

We first discuss information relevant to similarity coefficient-based fault localization, a better understanding of which shall lead to a better understanding of D* and the other fault localization techniques discussed herein.

2.1 Similarity Coefficient-based Fault Localization

Let us assume that we have a program P that consists of n executable statements[2] (in this paper we consider program components to be statements with the understanding that, without loss of generality, they could just as easily have been other components such as functions, blocks, predicates, etc.). Also consider that we have a test set T that consists of m different test cases. The program P is executed against each of these test cases and execution traces are collected, each of which records which statements in the program are covered (executed)[3] by the corresponding test case. The information on which test cases lead to successful executions (successful test cases) and which ones to failed executions (failed test cases) is also recorded. For the purposes of this paper and the discussions presented herein, a successful execution is one where the observed output of a program matches the expected output, and conversely a failed execution is one where the observed output of the program is different from that which is expected. This is consistent with the taxonomy defined in [4] where a failure is defined as an event that occurs when a delivered service deviates from correct (i.e. expected) service.

The collected data (as per above) can be represented as shown in Figure 1 which depicts a 'Coverage Matrix' and a 'Result Vector'. The coverage matrix and the result vector are binary such that – each entry (i,j) in the matrix is '1' if test case i covers statement j, and '0' if it does not; and each entry (i) in the result vector is '1' if test case i results in failure, and '0' if it is successful. Each row of the coverage matrix reveals which statements have been covered by the corresponding test case, and each column provides the coverage vector of the corresponding statement.

Intuitively, the 'closer' the execution pattern (i.e., the coverage vector) of a statement is to the failure pattern (result vector) of the test cases, the more likely the statement is to be faulty, and consequently the more suspicious the statement seems. By the same token, the farther (i.e. less similar or more dissimilar) the execution pattern of a statement is to the failure pattern, the less suspicious the statement appears to be. Similarity measures or coefficients[4] can be used to quantify this closeness/ similarity, and the degree of similarity can be interpreted as the suspiciousness of the statements.

Binary similarity measures are typically expressed by Operational Taxonomic Units (OTUs) [6,9] where each unit corresponds to a combination of matches/mismatches between the two vectors being evaluated for their similarity. Assuming two hypothetical vectors u and v, by way of notation – $\boldsymbol{a}$ is the number of features for which values of u and v are both 1; $\boldsymbol{b}$ is the number of features for which u has a value of 0 and v has a value of 1; $\boldsymbol{c}$ is the number of features for which u has a value of 1 and v has a value of 0; and $\boldsymbol{d}$ is the number of features where both u and v have a value of 0. The sum $\boldsymbol{a}+\boldsymbol{d}$ represents the total number of matches between the two vectors, and the sum $\boldsymbol{b}+\boldsymbol{c}$ represents the total number of mismatches between the two vectors. The aggregate sum $\boldsymbol{a}+\boldsymbol{b}+\boldsymbol{c}+\boldsymbol{d}$ represents the total number of features in either vector[5] which is typically denoted by n. This information can be expressed via a 2×2 contingency table, and in the context of fault localization can be represented for any statement as shown in Figure 2.

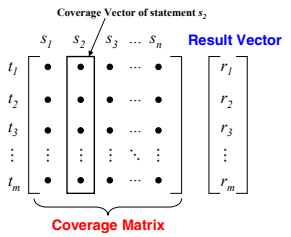

Figure 1. Data collected for fault localization

		Was the Statement Covered? Yes (1)	No (0)	SUM
Execution Result	Failure (1)	a (N_{CF})	b (N_{UF})	$a+b$ (N_F)
	Success (0)	c (N_{CS})	d (N_{US})	$c+d$ (N_S)
SUM		$a+c$ (N_C)	$b+d$ (N_U)	n

Figure 2. Contingency table in the context of fault localization

The quantities a, b, c and d (and their various combinations) in Figure 2 have been annotated with their equivalent fault localization-specific terms. This has been done primarily to enhance readability and to facilitate the discussions that are to follow. As per the contingency table, with respect to each statement the following terms are defined:

N_{CF}	number of failed test cases that covered the statement
N_{UF}	number of failed test cases that did not cover the statement
N_{CS}	number of successful test cases that covered the statement
N_{US}	number of successful test cases that did not cover the statement
N_C	total number of test cases that covered the statement
N_U	total number of test cases that did not cover the statement
N_S	total number of successful test cases
N_F	total number of failed test cases

Note that the quantities N_S and N_F are identical for all of the statements. Each fault localization technique discussed herein, including D*, consists of a suspiciousness function (or a ranking mechanism) that makes use of (some or all of) the above quantities to assign a value (suspiciousness) to a statement. This is illustrated via an example in Section 2.3.

2.2 The Construction of D*

Let us first list out our various intuitions regarding how the suspiciousness of a statement should be computed. Then, we construct a coefficient (i.e., the suspiciousness function) such that it realizes each of these intuitions appropriately.

1. The suspiciousness assigned to a statement should be directly proportional to the number of failed tests that cover it. The more a statement is executed by failed tests, the more suspicious it should be.

[2] Multi-line statements are combined so that they are counted as one during an execution, and code such as blank lines, comments, function and variable declarations are omitted from consideration as per the discussion in [15]. What we are then left with are executable statements. Hereafter, unless otherwise specified, we refer to executable statements simply as statements.

[3] In this paper, "a statement is *covered* by a test case" is used interchangeably with "a statement is *executed* by a test case"

[4] Since the coverage matrix and result vector are binary, we are essentially only concerned with binary similarity measures/coefficients.

[5] For the purposes of this paper, it is assumed that the vectors being evaluated for their mutual similarity are of equal size.

2. The suspiciousness assigned to a statement should be inversely proportional to the number of successful tests that cover it. The more a statement is executed by successful tests, the less suspicious it should be.
3. The suspiciousness assigned to a statement should be inversely proportional to the number of failed tests that do not cover it. If test cases fail without covering a particular statement, then that statement should be considered less suspicious.
4. Intuition 1 is the most sound and should carry a higher weight. We should assign greater importance to the information obtained from observing which statements are covered by failed tests than that obtained from observing which statements are covered by successful tests or which are not covered by failed tests.

Such intuitions are also followed by other fault localization techniques proposed in studies such as [15,26]. The first three are readily embodied by the Kulczynski coefficient which is expressed as $\frac{a}{b+c}$, in our case, $\frac{N_{CF}}{N_{UF}+N_{CS}}$. This however does not really lead to the realization of the fourth intuition and so is not suitable by itself.

In such a situation one might be tempted to multiply the numerator by (a number such as) 2 (thereby increasing the weight given to N_{CF} as compared to N_{UF} and N_{CS}), i.e., computing the suspiciousness for each statement as $\frac{2\times N_{CF}}{N_{UF}+N_{CS}}$. But such a modification is of no real benefit as we must remember that in the context of fault localization, we care not about the suspiciousness assigned to each statement from an absolute sense, but more from a relative sense; i.e., how the suspiciousness of one statement compares to that of another. As discussed before, the final goal is to produce a ranking consisting of statements arranged in decreasing order of suspiciousness. It then becomes clear that in terms of the ranking, scaling the suspiciousness assigned to each statement by a factor such as 2 will result in no change to the internal order of the statements in the ranking. Thus, the ranking produced by using $\frac{N_{CF}}{N_{UF}+N_{CS}}$ to compute suspiciousness will be the same as the one produced by using $\frac{2\times N_{CF}}{N_{UF}+N_{CS}}$ to compute suspiciousness. This is true of any non-trivial positive constant (excluding *zero* and *infinity*) and not just 2. Multiplying the numerator by such a number is of no immediate gain.

With this in mind, we make use of a coefficient such as $\frac{N^*_{CF}}{N_{UF}+N_{CS}}$ to compute the suspiciousness of each statement. This coefficient corresponding to D^* is thus a modification of the original Kulczynski coefficient, and is able to realize each of the 4 intuitions as prescribed. The value of * may vary. For discussion purposes, we use 2 in Section 2.3. However, in our case studies, the * has a value in the range from 2 to 50 with an increment of 0.5 (namely, 2.0, 2.5. 3.0, etc.) Note however that this is not the only way to realize said intuitions, and indeed many other suitable modifications may also be made. Additionally, we do not claim that the intuitions listed herein are the only intuitions that need to be followed in the context of fault localization. Instead, the usefulness of the proposed fault localization technique D^*, and its underlying intuitions, is demonstrated via rigorous case studies as presented in Section 3. Since it is not possible to theoretically prove if one fault localization technique shall always be more effective than another, such empirical validation is typically the norm for fault localization studies [1,7,15,17,20,26].

2.3 An Illustrative Example

We now walk through how D^2 (with * = 2) may be applied to generate a ranking of statements for fault localization. Consider the program in Figure 3 which presents a user with a choice on whether to compute the sum or average of two numbers. It has an error with regards to the sum computation on statement 5, i.e., instead of adding the two numbers, we accidentally subtract them. We also have a set of six test cases out of which four execute successfully (t_3, t_4, t_5 and t_6), while the other two (t_1 and t_2) result in failure. The coverage information for each test case is also shown where a black dot indicates that the corresponding statement (along the rows) has been covered, and the absence of a black dot indicates that the statement has not been covered by that test case.

		Coverage					
Stmt. #.	Program ***(P)***	t_1	t_2	t_3	t_4	t_5	t_6
1	read (a);	•	•	•	•	•	•
2	read (b);	•	•	•	•	•	•
3	read (choice);	•	•	•	•	•	•
4	if (choice == "sum")	•	•	•	•	•	•
5	result = a - b; //Correct: a + b;	•	•	•			
6	else if (choice == "average")				•	•	•
7	result = (a + b) / 2;				•	•	•
8	print (result);	•	•	•	•	•	•
Execution Result (0 = Successful / 1 = Failed)		1	1	0	0	0	0

Figure 3. A sample program with coverage and execution data

We begin by first collecting the statistics we need for D^2 (namely, N_{CF}, N_{CS} and N_{UF}) with respect to each statement as discussed in Section 2.2. These are shown in Table 1.

Table 1. Computation of suspiciousness using D^2

Stmt. #.	N_{CF}	N_{UF}	N_{CS}	Suspiciousness based on D^2 $N_{CF}\times N_{CF}/(N_{UF}+N_{CS})$
1	2	0	4	1
2	2	0	4	1
3	2	0	4	1
4	2	0	4	1
5	2	0	1	4
6	0	2	3	0
7	0	2	3	0
8	2	0	4	1

The suspiciousness for each statement is computed using D^2, which is shown in the rightmost column of Table 1. From the table we find that statement 5 is assigned the highest value (4), followed by statements 1, 2, 3, 4 and 8 (each tied with the same value of 1), and then statements 6 and 7 (with a suspiciousness value of 0 each). Thus, the highest position in the ranking would be assigned to statement 5, which in fact is the faulty statement as per our example.

This illustrates how D^* (where * may have different values) can be used to produce a ranking of program statements and lead programmers to faults automatically.

We are now ready to rigorously evaluate D^* against a large set of subject programs, while comparing its effectiveness against other fault localization techniques.

3. Case Studies

We first describe our experimental design in detail, and then move on to presenting the results of the case studies.

3.1 Subject Programs

Six different sets of programs – the *Siemens suite*, the *Unix suite*, *grep*, *gzip*, *make*, and *Ant* – have been used in our experiments, corresponding to a total of 21 different subject programs. An individual description of each set is as follows, with a comprehensive summary in Table 2.

The seven programs of the Siemens suite have been employed by many fault localization studies (such as [7,15,17] to name a few). All the correct and faulty versions (132 in all) of the programs (implemented in the C language) and test cases were downloaded from [21]. The Unix suite consists of ten Unix utility programs and since these programs have been so thoroughly used in industry and research alike, they can be a reliable and credible basis for our evaluations. The Unix suite has also been used in other studies such as [27].

Version 1.1.2 of the gzip program (which reduces the size of named files) was downloaded from [22]. Also downloaded from [22] were versions 2.2 of the *grep* program (which searches for a pattern in a file) and 3.76.1 of the *make* program (which manages building of executables and other products from source code), respectively. Each of these programs was downloaded with a set of test cases and faulty versions.

Since the above programs are all written in the C language, we also decided to use the Java-based *Ant* program (version 1.6 beta) downloaded from [22] along with test cases and faulty versions. For the *gzip*, *grep*, *make* and *Ant* programs, we created faulty versions using mutation-based fault injection, in addition to the ones that were downloaded. This was done to enlarge our data sets and because mutation-based faults have been shown to simulate realistic faults well and provide reliable and trustworthy results [3,8,17,18]. For grep, additional faults from [17] were also used. For all of our subject programs, any faulty versions that did not lead to at least one test case failure in our execution environment were excluded.

The subject programs vary dramatically both in terms of their size based on the LOC (lines of code) and functionality. The programs of the Siemens and Unix suites are small-sized (less than 1,000 lines of code), the gzip program is mid-sized (between 1,000 and 10,000 lines of code), the grep and make programs are large (between 10,000 and 20,000 lines of code), while the Ant program is of a very large size (greater than 75,000 lines of code). This allows us to evaluate across a very broad spectrum of programs and allows us to have more confidence in the ability to generalize our results.

3.2 Fault Localization Techniques Used in Comparisons

Since the proposed D^* technique has its origins in similarity coefficient-based analysis, we first compare it to other similarity coefficient-based techniques. In addition to the Kulczynski coefficient, we also compare with 11 other coefficients, forming a baker's dozen. A list of coefficients used along with their algebraic forms – based on the notation described in Section 2.1 – appear in Table 3.

Table 2. Summary of subject programs

	Program	Lines of Code	Number of faulty versions used	Number of test cases
Siemens suite	*print_tokens*	565	5	4130
	print_tokens2	510	10	4115
	schedule	412	9	2650
	schedule2	307	9	2710
	replace	563	32	5542
	tcas	173	41	1608
	tot_info	406	23	1052
Unix suite	*cal*	202	20	162
	checkeq	102	20	166
	col	308	30	156
	comm	167	12	186
	crypt	134	14	156
	look	170	14	193
	sort	913	21	997
	spline	338	13	700
	tr	137	11	870
	uniq	143	17	431
	gzip	6573	28	211
	grep	12653	19	470
	make	20014	31	793
	Ant	75333	23	871

Table 3. Similarity coefficient-based techniques used for comparison

	Coefficient	Algebraic Form
1	Kulczynski	$\frac{N_{CF}}{N_{UF}+N_{CS}}$
2	Simple-Matching	$\frac{N_{CF}+N_{US}}{N_{CF}+N_{CS}+N_{US}+N_{UF}}$
3	BraunBanquet	$\frac{N_{CF}}{max(N_{CF}+N_{CS},N_{CF}+N_{UF})}$
4	Dennis	$\frac{(N_{CF}\times N_{US})-(N_{CS}\times N_{UF})}{\sqrt{n\times(N_{CF}+N_{CS})\times(N_{CF}+N_{UF})}}$
5	Mountford	$\frac{N_{CF}}{0.5\times((N_{CF}\times N_{CS})+(N_{CF}\times N_{UF}))+(N_{CS}\times N_{UF})}$
6	Fossum	$\frac{n\times(N_{CF}-0.5)^2}{(N_{CF}+N_{CS})\times(N_{CF}+N_{UF})}$
7	Pearson	$\frac{n\times((N_{CF}\times N_{US})-(N_{CS}\times N_{UF}))^2}{N_C\times N_U\times N_S\times N_F}$
8	Gower	$\frac{N_{CF}+N_{US}}{\sqrt{N_F\times N_C\times N_U\times N_S}}$
9	Michael	$\frac{4\times((N_{CF}\times N_{US})-(N_{CS}\times N_{UF}))}{(N_{CF}+N_{US})^2+(N_{CS}+N_{UF})^2}$
10	Pierce	$\frac{(N_{CF}\times N_{UF})+(N_{UF}\times N_{CS})}{(N_{CF}\times N_{UF})+(2\times(N_{UF}\times N_{US}))+(N_{CS}\times N_{US})}$
11	Baroni-Urbani/ Buser	$\frac{\sqrt{(N_{CF}\times N_{US})}+N_{CF}}{\sqrt{(N_{CF}\times N_{US})}+N_{CF}+N_{CS}+N_{UF}}$
12	Tarwid	$\frac{(n\times N_{CF})-(N_F\times N_C)}{(n\times N_{CF})+(N_F\times N_C)}$

These coefficients have been used in different studies such as [6,25]. In this paper we treat each similarity coefficient as its own fault localization technique. For example, unless otherwise specified, when we refer to the Kulczynski technique (just 'Kulczynski' for short) we mean a fault localization technique that uses the Kulczynski coefficient for computing suspiciousness as described in Section 2.3.

As per this section, D^* has been compared to a total of 12 other similarity coefficient-based fault localization techniques (including Kulczynski in its original form). However, we point out that we also further compare D^* to other fault localization techniques and present the corresponding results in Section 4.

3.3 Evaluation Metrics/Criteria

For one fault localization technique to be considered more effective (better) than another, we must have suitable metrics of evaluation. Three metrics are used in this paper.

3.3.1 The *EXAM* Score

Renieris et al. [20] assign a score to every faulty version of each subject program, which is defined as the percentage of the program that need not be examined to find a faulty statement in the program or a faulty node in the corresponding program dependence graph. This score or effectiveness measure is later adopted by Cleve and Zeller in [7] and is defined as $1-\frac{|N|}{|PDG|}$ where N is the set of all nodes examined and PDG is the set of all nodes in the program dependence graph. Instead of the program dependence graph, the Tarantula fault localization technique [15] directly uses the program's source code. To make their effectiveness computation comparable to those of the program dependence graph, Jones et al. [15] consider only executable statements (see Footnote 2) to compute their score. A similar approach is used here.

However, while the authors of [15] define their score to be the percentage of code that need not be examined in order to find a fault, we feel it is more straightforward to present the percentage of code that *has to be examined* in order to find the fault. This modified score is hereafter referred to as *EXAM* and is defined as the percentage of executable statements that need to be examined until the first faulty statement is reached. Note that the two scores can be derived from one another.

A similar such modification is made by the authors of [17] where they define their effectiveness (T-score) as $T=\frac{|V_{examined}|}{|V|}\times 100\%,$ where $|V|$ is the size of the program dependence graph and $|V_{examined}|$ is the number of statements examined in a breadth first search before a faulty node is reached. Thus, the effectiveness of various fault localization techniques can be compared based on the *EXAM* score, and for any faulty program, if the *EXAM* score assigned by technique A is less than that of technique B, then A is considered to be more effective (better) than B (as relatively less code needs to be examined to locate faults).

3.3.2 Cumulative Number of Statements Examined

In addition to using the *EXAM* score, we also consider the total (cumulative) number of statements that need to be examined with respect to all faulty versions (of a subject program) to find faults. Formally, for any of the given subject programs, supposing we have n faulty versions, and $A(i)$ and $B(i)$ are the number of statements that must be examined to locate the fault in the i^{th} faulty version by techniques A and B, respectively, we can say that A is more effective than B, if $\sum_{i=1}^{n} A(i) < \sum_{i=1}^{n} B(i).$

3.3.3 Wilcoxon Signed-Rank Test

Additionally, in order to evaluate D^* based on sound statistics, we make use of the Wilcoxon signed-rank test (an alternative to the paired Student's t-test when a normal distribution of the population cannot be assumed) [19]. Since we aim to show that D^* is indeed more effective than other fault localization techniques, we evaluate the one-tailed alternate hypothesis that the other techniques require the examination of more statements than D^* (the null hypothesis being that the other techniques require the examination of a number of statements that is less than or equal to that required by D^*). Stated simply, the alternative hypothesis is that D^* will require the examination of fewer statements than the other techniques to locate faults, implying that D^* is more effective.

3.4 Data Collection

Each faulty version was executed against all its available test cases. Any deviation from the correct output was recorded as a failure, and a success was recorded if the test case output was identical between the correct and faulty versions. To collect coverage information for the C programs, we instrumented them using χSuds [28] which has the ability to record which statements of the source code are executed by a test case and how many times each of the statements is executed. In order for a statement to have been covered by a test case we required that it must have been executed by the test case at least once. In the case of Ant, Clover [12] was used to instrument the code and collect coverage data.

3.5 Results

Prior to presenting the results, we have one more concern that warrants discussion. We note that for D^* and all other techniques discussed herein, the suspiciousness value assigned to a statement may not always be unique. This means that two or more statements might be assigned the same suspiciousness and therefore be tied for the same position in the ranking.

Assume that a faulty statement and some correct statements share the same suspiciousness. Then, in the best case we examine the faulty statement first and in the worst case we examine it last and have to examine many correct statements before we discover the fault. This results in two different levels of effectiveness: the "best" and the "worst". In all our experiments we assume that for the "best" effectiveness the faulty statement is the first to be examined and for the "worst" effectiveness it is the last. During our evaluation of D^*, data is presented for these two levels of effectiveness, across all the evaluation metrics (criteria).

Table 4 presents the cumulative number of statements that need to be examined by each fault localization technique (as per Section 3.3.2) across each of the subject programs under study, for both best and worst cases. For example, we find that for the grep program, D^2 can locate faults in all faulty versions by requiring the examination of no more than 3023 statements in the best case and 4757 in the worst. Again with respect to the grep program, we find that the second best technique is Mountford which can locate all the faults by requiring the examination of no more than 3450 statements in the best case and 5189 in the worst. Note that these values correspond to the aggregate number

of statements that each technique requires the examination of to locate faults in *all* of the faulty versions per subject program (as opposed to per faulty version). The average number of statements to be examined per faulty version can easily be computed by dividing each entry in Table 4 by the number of faulty versions (corresponding to that program). For example on grep, in the best case D^2 requires the examination of about 159 (3023/19) statements (only 4.8% of the code) per faulty version and about 250 (4757/19) statements (7.56% of the code) in the worst.

Table 4. Total number of statements examined to locate faults for each of the subject programs under study (best and worst cases)

Fault Localization Technique	Best Case						Worst Case					
	Siemens	Unix	gzip	grep	make	Ant	Siemens	Unix	gzip	grep	make	Ant
D^2	**1754**	**1805**	**1220**	**3023**	**10287**	**672**	**2650**	**5226**	**3087**	**4757**	**16254**	**1184**
Kulcynski	2327	2358	1272	3458	10701	1557	3186	5779	3139	5192	16668	2069
Simple-Matching	6335	5545	9087	23806	41374	250414	7187	8977	10968	25606	48401	253631
BraunBanquet	2438	2767	1358	4114	11734	2196	3296	6187	3135	5847	17986	2698
Dennis	2206	2934	1960	5498	15016	1974	3074	6504	3737	8936	20755	2476
Mountford	1974	2183	1317	3450	11269	3298	2832	5644	3111	5189	17152	3818
Fossum	2230	2468	4547	15952	19567	150415	3126	5843	8701	21193	25036	150917
Pearson	3279	3581	1450	6894	17689	1188	4247	7221	3227	10796	23569	1690
Gower	6586	8630	26215	43428	128318	967307	7434	12027	27992	45262	134057	967809
Michael	1993	3713	2504	5027	14986	4502	2864	7283	4281	8501	20725	5004
Pierce	8072	11782	24065	16646	30568	322033	15299	23387	46753	60437	164856	1018725
Baroni-Urbani/Buser	3547	3189	1428	4902	12130	4693	4404	6605	3205	6635	17869	5195
Tarwid	2453	3399	3110	5793	16890	5964	3321	7883	5032	9517	23468	9935

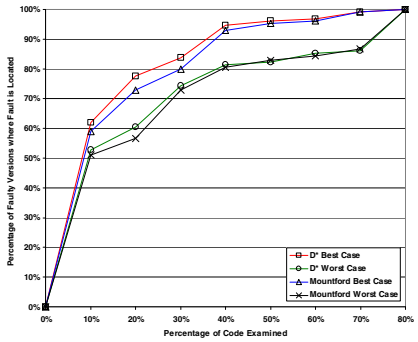

Figure 4. *EXAM* score-based comparison on the Siemens suite

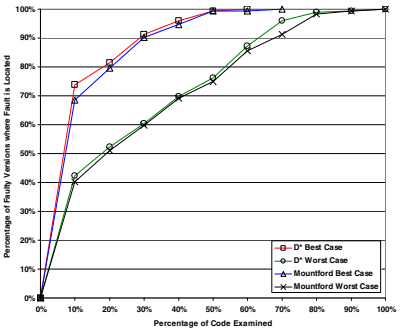

Figure 5. *EXAM* score-based comparison on the Unix suite

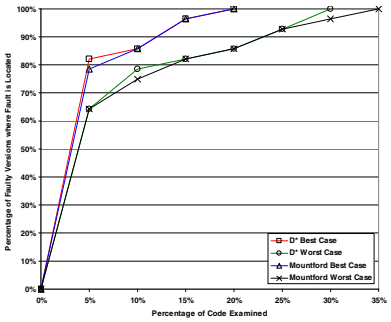

Figure 6. *EXAM* score-based comparison on the gzip program

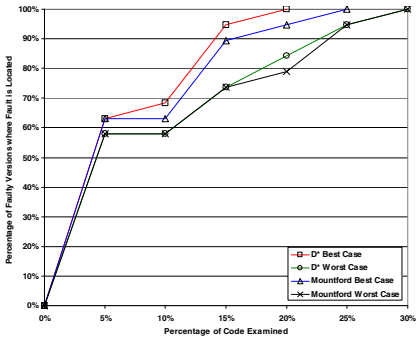

Figure 7. *EXAM* score-based comparison on the grep program

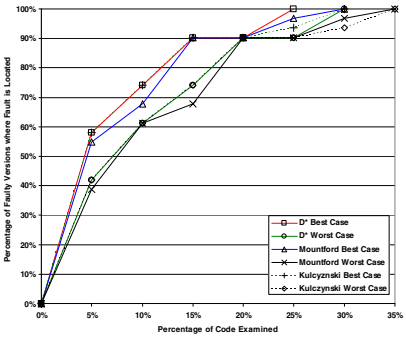

Figure 8. *EXAM* score-based comparison on the make program

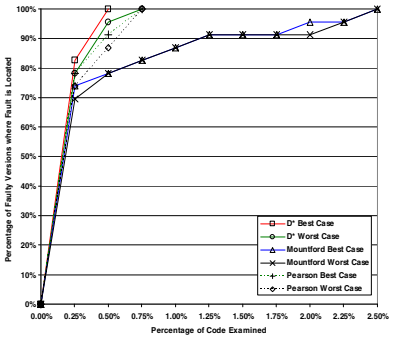

Figure 9. *EXAM* score-based comparison on the Ant program

From Table 4 we observe that regardless of whether the best or worst case is considered, D^2 is always the most effective of the similarity coefficient-based fault localization techniques across each of the subject programs. This justifies not only the modification of Kulczynski to D^2 (as D^2 performs better than Kulczynski) but also the use of D^2 as a highly effective fault localization technique. The effectiveness of D^* improves as the value of the * increases until it levels off (see Section 4).

We re-emphasize that D^2 is not just the most effective technique on *most* of the subject programs under study, but rather on *all* subject programs. This consistent high performance with respect to D^2 deserves attention as the other techniques are not just less effective than D^2, but are also less consistent. For example, from an overall perspective (considering all of the subject programs) while Mountford seems to be the second best technique, it is not the second best throughout all subject programs. Consider that on the *make* program, Kulczynski is the second most effective technique (irrespective of best or worst case), while on the *Ant* program, Pearson is the second most effective (irrespective of best or worst case). However, D^2 consistently performs better than all other techniques, and this is true on all subject programs.

Table 5. Confidence with which it can be claimed that D^* is more effective than other techniques (Best and Worst Cases)

Fault Localization Technique	Best Case						Worst Case					
	Siemens	Unix	gzip	grep	make	Ant	Siemens	Unix	gzip	grep	make	Ant
Kulcynski	99.99%	99.99%	93.75%	99.60%	98.43%	98.43%	99.99%	99.99%	93.75%	99.60%	98.43%	98.43%
Simple-Matching	100%	100%	99.80%	99.98%	99.99%	99.90%	100%	100%	97.60%	99.99%	99.99%	99.80%
BraunBanquet	99.99%	100%	99.80%	99.95%	99.99%	99.80%	99.99%	99.99%	71.43%	99.92%	99.83%	99.21%
Dennis	99.99%	100%	99.99%	99.80%	99.99%	99.80%	99.99%	100%	94.20%	99.80%	99.99%	99.21%
Mountford	99.99%	99.99%	99.21%	99.60%	99.95%	99.90%	99.99%	99.99%	73.82%	99.90%	99.01%	99.80%
Fossum	100%	99.99%	99.21%	99.90%	99.80%	99.21%	100%	99.99%	99.62%	99.97%	93.99%	96.87%
Pearson	100%	99.99%	99.21%	99.95%	99.99%	99.21%	100%	99.99%	70.87%	99.99%	99.99%	96.87%
Gower	100%	100%	99.99%	99.99%	99.99%	99.99%	100%	100%	99.99%	99.99%	99.99%	99.99%
Michael	99.68%	99.99%	99.99%	99.99%	99.99%	99.97%	99.54%	99.99%	99.99%	99.99%	99.99%	99.97%
Pierce	100%	100%	99.99%	99.99%	99.99%	99.99%	100%	100%	99.99%	99.99%	99.99%	99.99%
Buser	99.99%	100%	99.80%	99.95%	99.99%	99.80%	99.99%	100%	74.42%	99.92%	99.90%	98.82%
Tarwid	99.99%	99.99%	99.99%	99.98%	99.99%	99.99%	99.99%	100%	99.99%	99.99%	99.99%	99.99%

Figures 4 through 9 present the evaluation of D^2 with respect to the *EXAM* score on the Siemens suite, Unix suite, gzip, grep, make and Ant programs. In each figure, the *x*-axis represents the percentage of code (executable statements) examined while the *y*-axis represents the number of faulty versions where faults are located by the examination of an amount of code less than or equal to the corresponding value on the *x*-axis. For example, based on Figure 4, we find that on the Siemens suite, by examining less than 10% of the code, D^2 can locate faults for 62.02% of the faulty versions in the best case and 52.71% in the worst. In contrast, by examining less than 10% of the code, Mountford can only locate 58.91% of the faults in the best case and 51.16% in the worst. The comparisons in these figures are provided only with respect to Mountford purely in the interests of clarity and readability[6].

Mountford was chosen as based on Table 4, it provides the second best performance on the most subject programs. An exception is made for the *make* and *Ant* programs (Figures 8 and 9) where the curves corresponding to the Kulczynski (Figure 8) and Pearson (Figure 9) techniques are also included, as these techniques are more effective than Mountford. We emphasize that while the curves with respect to all of the other similarity coefficient-based techniques have not been presented in this paper, the corresponding curves were nevertheless generated, and with respect to the *EXAM* score, D^2 performs better than all of the other similarity coefficient-based techniques evaluated (which is consistent with our observations from Table 4).[7]

The figures corroborate our observations from Table 4, in that the best case of D^2 is always better than the best case of each of the competing techniques, and the worst case of D^2 is always better than the worst case of the competing techniques. In fact, often the worst case of D^2 is also better than or at least comparable to the best case of the competing techniques, as is clearly seen in the case of the *Ant* program (refer to Figure 9). This clearly indicates the superiority of D^2 over the other techniques when evaluated using the *EXAM* score.

With respect to our third metric of evaluation, Table 5 presents data comparing D^2 to the other techniques using the Wilcoxon signed-rank test. Each entry in the table gives the confidence with which the alternative hypothesis (that D^2 requires the examination of fewer statements than the other technique to find faults, thereby making it more effective) can be accepted with respect to each of the subject programs for both the best and worst cases. To take an example, it can be said with 99.80% confidence that D^2 is more effective than BraunBanquet on the *Ant* program in the best case and with 99.21% confidence (again on *Ant*) in the worst case.

Let us now modify our alternative hypothesis to consider equalities. Thus, we now consider the alternative that D^2 requires the examination of a lesser or equal number of statements to find faults than other techniques, i.e., D^2 is more effective than or at least as effective as the other techniques. We present data with respect to this modified alternative hypothesis for the subject programs and techniques where the confidence with respect to the original alternative is lower than 99% (referring to Table 5) in Table 6. Evaluating with respect to the other techniques and subject programs is not necessary as the corresponding confidence values can only be raised (but not lowered) with respect to the new alternative hypothesis, and they are already beyond the 99% level with respect to the original alternative hypothesis.

Table 6. Confidence with which it can be claimed that D^2 is more effective than or at least as effective as other techniques (Best and Worst Cases)

Fault Localization Technique	Best Case			Worst Case		
	gzip	make	Ant	gzip	make	Ant
Kulcynski	100%	100%	100%	100%	100%	100%
Simple-Matching	100%	100%	100%	99.94%	100%	99.90%
BraunBanquet	100%	100%	100%	99.14%	99.85%	99.61%
Dennis	100%	100%	100%	99.43%	100%	99.61%
Mountford	100%	100%	100%	95.78%	99.13%	99.90%
Fossum	100%	100%	100%	99.67%	99.78%	99.44%
Pearson	100%	100%	100%	92.19%	100%	98.44%
Buser	100%	100%	100%	95.42%	99.91%	99.22%

From Table 6 we find that with respect to the alternative hypothesis that D^2 is more effective than or at least as effective as the other techniques, we can accept this claim with 100% confidence (because the corresponding p-values are trivially small) when the best case is considered. As for the worst case, our alternative hypothesis can always be accepted with a minimum of 95% confidence except for Pearson on the *gzip* program, where the hypothesis is accepted at 92.19% which is in no way a low confidence. If we also consider the fact that D^2 is better than Pearson in terms of the *EXAM* score

[6] Each technique evaluated would have two curves corresponding to the best and worst cases of that technique, respectively. Including each of the 13 techniques (12 competing techniques plus D^2) would have meant 26 different curves in each figure, which makes it very difficult, if not entirely impossible, to decipher any useful information from the figures.

[7] Readers interested in obtaining the complete set of curves (i.e., figures with all of the curves) may contact the authors for copies.

(an example of which is in Figure 9) and the total number of statements that need to be examined to find all the faults, we can comfortably conclude that D^2 is much better than Pearson as a fault localization technique. In summary, our case studies clearly show that D^2 is much more effective than the other similarity coefficient-based techniques at fault localization.

4. Effectiveness of D^* with different values for the *

The impact of different values of * on the effectiveness of D* was examined. The range used is from 2 to 50 with an increment of 0.5 (namely, * = 2.0, 2.5, 3.0, …, 50.0). From Figure 10 we observe that the total number of statements examined to locate all the bugs in the Siemens suite declines as the value of the * increases from 2 to 33, and after that the number remains almost the same. For example, 1754 statements need to be examined for D^2 in the best case, while the number decreases to 1526 for D^3, and further down to 1400 for D^6. That is, the effectiveness of D^* increases as the value of * increases until it levels off. A similar observation was also made for other subject programs.

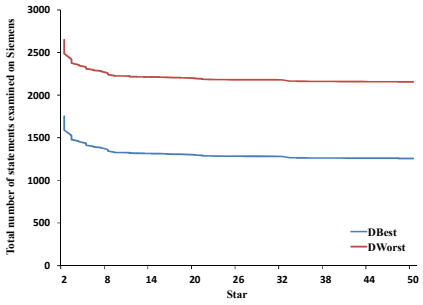

Figure 10. Effectiveness of D^* with respect to different values of * on the Siemens suite

5. D^* Versus Other Fault Localization Techniques

So far D^* has been compared to other well-known similarity coefficients in the context of fault localization, and results indicate that D^* is the most effective. However, various techniques have been reported in other studies that also claim to be effective at fault localization and so it is important to compare D^* to such techniques as well. Since it is physically impossible to compare D^* to all techniques, we select a representative set of techniques – Tarantula [15], Ochiai [1] and Heuristic III [26] – and compare D^* to them.

The Tarantula fault localization technique [15] assigns a suspiciousness value to each statement as per the formula X/X+Y, where based on the notation defined in Section 2.1, $X=(N_{CF}/N_F)$ and $Y=(N_{CS}/N_S)$. Tarantula has been shown to be more effective than several other fault localization techniques such as set union, set intersection, nearest neighbor, and cause transitions on the Siemens suite [15]. In [1], Abreu et al. evaluate the use of the Ochiai coefficient in the context of software fault localization. Ochiai assigns a suspiciousness value to each statement using the following formula

$$\frac{N_{CF}}{\sqrt{N_F \times (N_{CF} + N_{CS})}}$$

Wong et al. examined how each additional failed (or successful) test case can help locate program faults [26]. They concluded that the contribution of the first failed test is larger than or equal to that of the second failed test, which is larger than or equal to that of the third failed test, and so on. The same applies to successful tests. The suspiciousness of each statement is computed as $[(1.0)\times n_{F,1}+(0.1)\times n_{F,2}+(0.01)\times n_{F,3}] -[(1.0)\times n_{s,1}+(0.1)\times n_{s,2}+\alpha\times\chi_{F/S}\times n_{s,3}]$, where $n_{F,i}$ and $n_{S,i}$ are the number of failed and successful tests in the i^{th} group, and $\chi_{F/S}$ is the ratio of the total number of failed to the total number of successful tests with respect to a given bug. The technique is named H3b when $\alpha = 0.001$ and H3c with $\alpha = 0.0001$.

Figure 11 presents data on the total number of statements to be examined by using D^*, H3b, H3c, Tarantula, and Ochiai, to locate faults in all the faulty versions of each subject program for both the best and worst cases. As seen from this table, D^* (with an appropriate value for the *) is better than other techniques irrespective of subject program, or whether the best or worst case is considered. For example, with respect to Unix, D^* is more effective than others when * is greater than or equal to 4. The same applies to gzip and Ant when * is greater than or equal to 2, and for Siemens, grep and make when * is greater than or equal to 7, 8, and 20, respectively. Moreover, in some cases the worst case of D^* is even better than the best cases of other techniques. For example, for the grep and Ant programs, the worst case of D^2 is also better than the best case of Tarantula.

	Best Case					
	Unix	Siemens	grep	gzip	make	Ant
D^2	1805	1754	3023	1220	10287	672
D^3	1667	1526	2946	1088	10257	368
D^4	1594	1460	2833	1087	10022	293
D^5	1507	1435	2762	1085	10022	228
D^*		1386 (*=7)	2693 (*=8)		8529 (*=20)	
H3b	1701	1439	3019	1535	10817	1358
H3c	1655	1396	2702	1535	8553	1320
Tarantula	3394	2453	5793	3110	16890	5964
Ochiai	1906	1796	3092	1270	10305	887

(a) Best case comparison

	Worst Case					
	Unix	Siemens	grep	gzip	make	Ant
D^2	5226	2650	4757	3087	16254	1184
D^3	5088	2422	4680	2955	16224	880
D^4	5015	2356	4567	2954	15989	805
D^5	4928	2331	4496	2952	15989	740
D^*		2284 (*=7)	4427 (*=8)		14219 (*=25)	
H3b	5072	2335	4752	3313	16556	1860
H3c	5026	2292	4435	3312	14272	1882
Tarantula	7704	3311	7812	5032	23468	9935
Ochiai	5322	2692	4825	3047	16044	1389

(b) Worst case comparison

Figure 11. D^* versus H3b, H3c, Tarantula and Ochiai: Total number of statements examined to locate all the faults. The cell with a black background gives the smallest value of * such that D^* outperforms the others.

The Wilcoxon signed-rank test was also conducted. The confidence to accept the alternative hypothesis (D^* is more effective than the other techniques) ranges from 93 to 100%, regardless of the subject programs, or the best or worse case.

6. Programs with Multiple Faults

Thus far, the evaluation of D^* has been with respect to programs that have exactly one fault in them. In this section, we discuss and demonstrate how D^* may very easily be applied to programs with multiple faults in them as well.

6.1 The Expense Score-based Approach

For programs with multiple faults, the authors of [30] define an evaluation metric, *Expense*, corresponding to the % of code

that must be examined to locate the first fault as they argue that this is the fault that programmers will begin to fix. We note that the *Expense* score, though defined in a multi-fault setting, is very similar to the *EXAM* score used in this paper.

Thus, D^* can also be applied to and evaluated on programs with multiple faults in such a manner, and in accordance we conduct a comparison between D^* (for * equals 2 and 3 in this case) and other fault localization techniques using this strategy. As per [30] all the techniques are evaluated on the basis of the "expense" required to find only the first fault in the ranking. Note that since the rankings based on D^* and the other techniques may vary considerably, it is not necessary that the first fault located by one technique be the same as the first fault located by another technique.

Multi-fault versions of the subject programs are created via a combination of single-fault versions. The programs of the Siemens suite have been used for the purposes of this comparison as there are many single-fault versions available which can be combined in a variety of ways to produce many different multi-fault versions. A total of 75 such programs are created based on combinations of the single-fault programs of the Siemens suite, and they range from faulty versions with 2 faults in them to those with 5 faults in them. To evaluate the effectiveness, we compare D^* against Mountford and Ochiai, as they provide the closest performance to D^* when compared to the other techniques evaluated. We also compare D^* to Tarantula as it is used in [30].

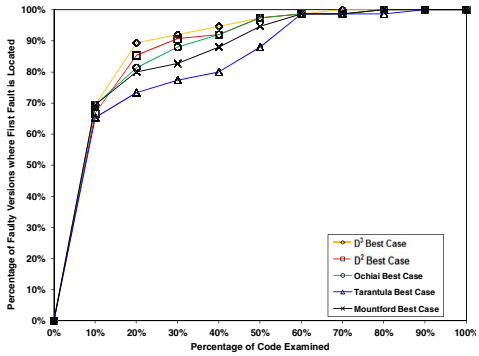

Figure 12. Best case comparison for the 75 multi-fault versions of the Siemens suite using the Expense/*EXAM* score

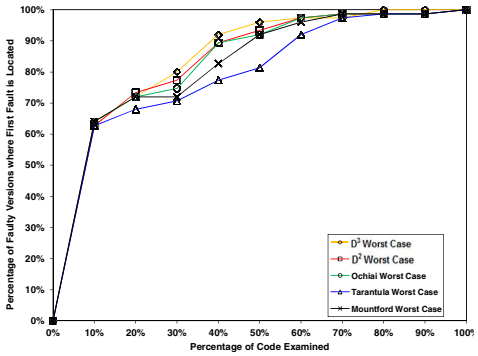

Figure 13. Worst case comparison for the 75 multi-fault versions of the Siemens suite using the Expense/*EXAM* score

Table 7 Total number of statements examined for the 75 multi-fault versions of the Siemens suite by D^*, Tarantula, Ochiai and Mountford

Fault Localization Technique	Best Case	Worst Case
D^2	932	1374
D^3	884	1327
Tarantula	1451	1851
Ochiai	976	1385
Mountford	1135	1546

Figure 12 gives the best case comparison using the *Expense* score, while Figure 13 presents the worst. We see that D^3 generally performs better than D^2, which is better than others.

Data in Table 7 gives the total number of statements that need to be examined to find the first fault across all 75 faulty versions. We observe that D^3 is more effective than D^2, which is more effective than the other techniques, regardless of best or worst case. With respect to the Wilcoxon signed-rank test, due to the nature of the Expense score, the alternative hypothesis in this case would be that D^3 is more effective at *finding the first fault* in a multi-fault program than the technique under comparison. The corresponding confidence levels at which the alternative hypothesis can be accepted is presented in Table 8. The data suggests that even for multi-fault programs, it can be confidently claimed that D^3 is more effective than Tarantula, Mountford, and Ochiai, irrespective of best or worst case. Together with the curves in Figure 12 and Figure 13, and the data in Table 7, there is strong evidence for the superiority of D^* (with an appropriate value for the *) over all the techniques compared in this paper.

Table 8. Confidence with which it can be claimed that D^3 is more effective than other techniques (Best and Worst Cases) based on the *Expense* Score

Fault Localization Technique	Best Case	Worst Case
D^2	98.67%	98.66%
Tarantula	99.98%	99.92%
Mountford	99.70%	99.09%
Ochiai	98.41%	96.31%

7. Related Work

We now overview fault localization studies in addition to the ones already discussed in this paper, directing readers interested in further details to the accompanying references.

In [20], a nearest neighbor debugging technique is proposed by Renieres and Reiss that contrasts a failed test with another successful test (most similar to the failed one in terms of the 'distance' between them). If a bug is in the difference set between the failed execution and its most similar successful execution, it is located. For a bug not in the difference set, the technique continues by constructing a program dependence graph and checking adjacent un-checked nodes in the graph step by step until the bug is located.

Also proposed in [20] are the set union and set intersection techniques. The former computes the set difference between the program spectrum of a failed test and the union spectra of a set of successful tests. It focuses on the source code that is executed by the failed test but not by any of the successful tests. The latter is based on the set difference between the intersection spectra of successful tests and the spectrum of the failed test. It focuses on statements that are executed by all successful tests but not by the failed test case.

Cleve and Zeller [7] report a program state-based debugging technique, cause transition, to identify the locations and times where a cause of failure changes from one variable to another. This is an extension of their earlier work with delta debugging [31,32]. An algorithm named *cts* is proposed to quickly locate cause transitions in a program execution. A potential problem of this technique is that its cost is relatively high; there may exist thousands of states in a program execution, and delta debugging at each matching point requires additional test runs to narrow down the causes.

As per [15], Tarantula performs better than techniques such as nearest neighbor, set union, set intersection, and cause

transitions on the Siemens suite. Since D^* is significantly more effective than Tarantula, D^* is also expected to outperform the above techniques. In [5] a way to optimize Tarantula is presented which selects a subset of available test cases (as opposed to the entire test set) to maximize the number of basic blocks covered. In [14] Tarantula is used to explore how test cases can be clustered to *debug in parallel* (simultaneously debug multiple faults in a program). Zhang et al. focus on the propagation of infected program states in [33].

In [23] the original Tarantula technique [15] is replaced by Ochiai [1] (which has also been shown to be less effective than D^*) to evaluate the quality of fault localization with respect to multiple coverage types, namely statements, branches and data dependencies. For the purposes of this paper, only statement-based coverage is used as an input to the various fault localization techniques, and the effects of using multiple coverage types with D^* is deferred to future study.

8. Conclusions and Future Work

In this paper we propose a technique named D^* to automatically lead developers to the locations of faults in programs. Via an evaluation across 21 different programs, D^* is shown to be more effective than 16 other contemporary fault localization techniques (12 similarity coefficient-based, the Tarantula, Ochiai, H3b and H3c techniques). Furthermore, the superior effectiveness of D^* is not limited to programs containing just one fault; based on our experiments it extends equally well to programs with multiple faults.

In the future we wish to apply D^* to an ongoing large scale industrial project in collaboration with our industry partners. Since D^* is based on a modification of the Kulczynski coefficient, we also wish to investigate whether D^* can be further evolved (using logical modifications) to be even more effective at locating faults than the current form.

References:

1. R. Abreu, P. Zoeteweij, R. Golsteijn, and A.J.C. van Gemund, "A Practical Evaluation of Spectrum-based Fault Localization", *Journal of Systems and Software*, 82(11):1780 - 1792, 2009
2. H. Agrawal, J. R. Horgan, S. London, and W. E. Wong, "Fault localization using execution slices and dataflow tests," in *Proc. of the 6th International Symposium on Software Reliability Engineering,* pp. 143-15, Toulouse, France, October, 1995.
3. J.H. Andrews, L.C. Briand and Y. Labiche, "Is mutation an appropriate tool for testing experiments?" in *Proc. of the 27th International Conference on Software Engineering*, pp. 402-411, St. Louis, Missouri, USA, May, 2005.
4. A. Avizienis, J. C. Laprie, B. Randell, and C. Landwehr, "Basic concepts and taxonomy of dependable and secure computing," *IEEE Transactions on Dependable and Secure Computing*, 1(1):11-33, 2004
5. B. Baudry, F. Fleurey, and Y. le Traon, "Improving test suites for efficient fault localization," in *Proc. of the International Conference on Software Engineering*, pp. 82-91, Shanghai, China, May 2006.
6. S. Choi, S. Cha, C. C. Tappert, "A Survey of Binary Similarity and Distance Measures", *Journal of Systemics, Cybernetics and Informatics*, 8(1): 43-48, Jan 2010
7. H. Cleve and A. Zeller, "Locating causes of program failures," in *Proc. of the 27th International Conference on Software Engineering*, pp. 342-351, St. Louis, Missouri, USA, May, 2005.
8. H. Do and G. Rothermel, "On the use of mutation faults in empirical assessments of test case prioritization techniques" *IEEE Transactions on Software Engineering*, 32(9): 733-752, September 2006.
9. G. Dunn and B.S Everitt, An Introduction to Mathematical Taxonomy, Cambridge University Press, 1982
10. S. Fitzgerald, R. McCauley, B. Hanks, L. Murphy, B. Simon, and C. Zander, "Debugging from the Student Perspective," in *IEEE Transactions on Education*, 53(3): 390-396, August 2010
11. A.L. Goel, "Software reliability models: Assumptions, limitations and applicability," *IEEE Transactions on Software Engineering*, 11(12): 1411-1423, Dec. 1985
12. http://www.atlassian.com/software/clover/ (Clover: A code coverage analysis tool for Java)
13. M. Jiang, M. A. Munawar, T. Reidemeister, and P. Ward, "Efficient fault detection and diagnosis in complex software systems with information-theoretic monitoring," *IEEE Transactions on Dependable and Secure Computing*, 8(4): 510-522, July/August 2011.
14. J. A. Jones, J. Bowring, and M. J. Harrold, "Debugging in parallel," in *Proc. of the 2007 International Symposium on Software Testing and Analysis*, pp. 16-26, London, UK, July, 2007.
15. J. A. Jones and M. J. Harrold, "Empirical evaluation of the Tarantula automatic fault-localization technique," in *Proc. of the 20th IEEE/ACM Conference on Automated Software Engineering*, pp. 273-282, Long Beach, California, USA, December, 2005.
16. B. Liblit, M. Naik, A. X. Zheng, A. Aiken, and M. I. Jordan, "Scalable statistical bug isolation," in *Proc. of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pp. 15-26, Chicago, Illinois, USA, June, 2005.
17. C. Liu, L. Fei, X. Yan, J. Han, and S. P. Midkiff, "Statistical debugging: a hypothesis testing-based approach," *IEEE Transactions on Software Engineering*, 32(10):831-848, October, 2006.
18. A. S. Namin, J. H. Andrews and Y. Labiche, "Using mutation analysis for assessing and comparing testing coverage criteria," *IEEE Transactions on Software Engineering*, 32(8):608-624, August 2006
19. R. Lyman Ott, "An introduction to statistical methods and data analysis (Fourth Edition)," Duxbury Press, Wadsworth Inc., 1993
20. M. Renieres and S. P. Reiss, "Fault localization with nearest neighbor queries," in *Proc. of the 18th International Conference on Automated Software Engineering*, pp. 30-39, Montreal, Canada, October, 2003.
21. The Siemens Suite, www-static.cc.gatech.edu/aristotle/Tools/subjects
22. The Software Infrastructure Repository. http://sir.unl.edu/portal/index.html
23. R. Santelices, J.A. Jones, Y. Yu, and M.J. Harrold, "Lighweight fault-localization using multiple coverage types," in *Proc. of the 31st International Conference on Software Engineering*, pp. 56-66, Vancouver, Canada, May 2009
24. I. Vessey, "Expertise in debugging computer programs," in *International Journal of Man Machine Studies* :Process Analysis, 23(5):459-494, 1985
25. P. Willett, "Similarity-based approaches to virtual screening," Biochemical Society Transactions, 31(3):603-606, Jun 2003
26. W. E. Wong, V. Debroy and B. Choi, "A Family of Code Coverage-based Heuristics for Effective Fault Localization," *Journal of Systems and Software*, 83(2):188-208, February 2010
27. W. E. Wong**,** J. R. Horgan, S. London, and A. P. Mathur, "Effect of Test Set Minimization on Fault Detection Effectiveness," *Software-Practice and Experience*, 28(4):347-369, April 1998
28. χSuds User's Manual, Telcordia Technologies, 1998.
29. M. Xie and B. Yang, "A study on the effect of imperfect debugging on software development cost," *IEEE Transactions on Software Engineering*, 29(5): 471-473, May 2003
30. Y. Yu, J. A. Jones and M. J. Harrold, "An empirical study on the effects of test-suite reduction on fault localization," in *Proc. of the International Conference on Software Engineering* (ICSE), pp. 201-210, Leipzig, Germany, May 2008
31. A. Zeller, "Isolating cause-effect chains from computer programs," in *Proc.s of the 10th ACM SIGSOFT symposium on Foundations of software engineering*, pp. 1-10, Charleston, South Carolina, USA, November, 2002.
32. A. Zeller and R. Hildebrandt, "Simplifying and isolating failure-inducing input," *IEEE Transactions on Software Engineering*, 28(2):183-200, February, 2002.
33. Z. Zhang, W.K. Chan, T.H. Tse, B. Jiang, and X. Wang, "Capturing propagation of infected program states," in *Proc. of the 7th joint meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE), pp. 43-52, Amsterdam, The Netherlands, August 2009*