The DStar Method for Effective Software Fault Localization

W. Eric Wong, Vidroha Debroy, Ruizhi Gao, and Yihao Li

***Abstract*—Effective debugging is crucial to producing reliable 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^*) is proposed which can suggest suspicious locations for fault localization automatically without requiring any prior information on program structure or semantics. D^* is evaluated across 24 programs, and is compared to 38 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. An empirical evaluation is also conducted to illustrate how the effectiveness of D^* increases as the exponent * grows, and then levels off when the exponent * exceeds a critical value. Discussions are presented to support such observations.**

***Index Terms*—Testing, debugging, software fault localization, EXAM score.**

ACRONYMS:

Crosstab	cross tabulation
RBF	radial basis function
D^*	DStar
H3b	Heuristic III(b)
H3c	Heuristic III(c)
LOC	lines of code

NOTATION:

P	a generic program
N_{CF}	number of failed test cases that cover the statement
N_{UF}	number of failed test cases that do not cover the statement
N_{CS}	number of successful test cases that cover the statement
N_{US}	number of successful test cases that do not cover the statement
N_C	total number of test cases that cover the statement
N_U	total number of test cases that do not cover the statement
N_S	total number of successful test cases
N_F	total number of failed test cases
t_i	the ith test case

Manuscript received September 16, 2012; revised February 19, 2013; accepted May 20, 2013. Date of publication November 01, 2013; date of current version February 27, 2014. Associate Editor: S. Shieh.

The authors are with the Department of Computer Science, University of Texas at Dallas, Richardson. TX 75080 USA (e-mail: ewong@utdallas.edu).

Color versions of one or more of the figures in this paper are available online at http://ieeexplore.ieee.org.

Digital Object Identifier 10.1109/TR.2013.2285319

I. INTRODUCTION

IT is a well-recognized fact that manual debugging is not just time-consuming, tedious, and expensive, but also quite error-prone [16], [25], [46]. Thus, approaches that can help automate the debugging process, thereby making software more reliable 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 [35]. 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 [14]. 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 speed reduces the amount of downtime a system may incur, thereby improving availability [21]. Such realizations have paved the way for the proposal and development of a number of different fault localization techniques over recent years, each of which tries to identify the locations of faults in one way or another [1], [3], [8], [23]–[25], [31], [39], [50].

A useful 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 [7], [37] 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

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

0018-9529 © 2013 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission.
See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.

one and the same [7], [29], [37]), and thus the modified version of the coefficient may already exist. 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 nine different sets of programs (*Siemens suite*, *Unix suite*, *gzip*, *grep*, *make*, *Ant*, *space*, *flex*, and *sed*) corresponding to a total of 24 different subject programs (Table II in Section 3.1), 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 original Kulczynski coefficient. We formally define what we mean by *better* in Section 3.3. The use of D* as a fault localization technique is further supported by its better performance compared to 38 other techniques (each described in detail subsequently in this paper) including 31 similarity coefficient-based techniques (Section 3), and 7 other contemporary techniques (Section 5). The impact of different values of * on the effectiveness of D* is also examined, followed by a detailed analysis to validate our observations.

The major contributions of this paper can be summarized as follows.

1. A fault localization technique named D* is proposed.
2. The effectiveness of D* is evaluated across 24 programs, and compared to 38 different fault localization techniques. It is demonstrated that D* is better (or more effective) at fault localization than the other 38 techniques.
3. The variation of the effectiveness of D* is investigated with respect to different values of *. An analysis is conducted to explain why the effectiveness increases as * grows, and eventually levels off when * exceeds a critical value. A discussion of the upper bound on the effectiveness of D* is presented.

The rest of the paper is organized as follows. Section 2 overviews similarity coefficient-based fault localization, defines D*, and provides an illustrative example. Section 3 presents our case studies, reporting details on the experimental design, data collection, and more, before moving on to the results. Section 4 shows how the effectiveness of D* improves and levels off as the value of * 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 discusses relevant issues as well as threats to validity. Section 8 describes work related to the subject presented in this paper. Finally, we provide our conclusions and plans for future work in Section 9.

II. 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.

A. Similarity Coefficient-Based Fault Localization

Let us assume that we have a program P that consists of n statements. 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 comprises 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)[2] by the corresponding test case. The information on which test cases lead to successful executions (successful test cases) and which ones lead 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 indicator is consistent with the taxonomy defined in [5] 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 the discussion above) can be represented as shown in Fig. 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. Thus, 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 more similar 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[3] can be used to quantify this closeness or 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 [7], [12], where each unit corresponds to a combination of matches and 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.

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

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

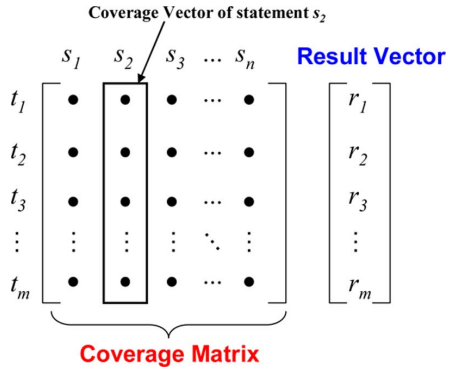

Fig. 1. Data collected for fault localization.

		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

Fig. 2. Contingency table in the context of fault localization.

The aggregate sum $\boldsymbol{a+b+c+d}$ represents the total number of features in either vector,[4] 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 Fig. 2.

Note that the quantities a, b, c, and d (and their various combinations) in Fig. 2 have been annotated with their equivalent fault localization-specific terms. This annotation has been done to enhance readability, and to facilitate the discussions that are to follow. As per the contingency table in Fig. 2, the notations with respect to each statement are defined in the Notation section at the beginning of the paper. 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) these quantities to assign a value (suspiciousness) to a statement. This approach is illustrated via an example in Section 2.3.

B. The Construction of D*

Let us first list 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 frequently a statement is executed by failed tests, the more suspicious it should be.
2. The suspiciousness assigned to a statement should be inversely proportional to the number of successful tests that cover it. The more frequently 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 [23], [39]. The first three are readily embodied by the Kulczynski coefficient which is expressed as $a/(b+c)$, or in our case as $N_{CF}/(N_{UF}+N_{CS})$. This approach 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 some constant factor such as 2 to increase the weight given to N_{CF} as compared to N_{UF} and N_{CS}. But such a modification is of no real benefit in the context of fault localization, as we only care about using the relative suspiciousness of one statement compared to another to produce a ranking of statements arranged in decreasing order of suspiciousness. Scaling the suspiciousness assigned to each statement by a constant factor (excluding *zero* and *infinity*) will result in no change to the internal order of the statements.

We therefore make use of a coefficient such as $(N_{CF})^*/(N_{UF}+N_{CS})$, where the value of * is greater than or equal to 1, 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 set * to either 2 or 3 in Section 2.3. However, in our case studies, * 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 that this approach 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 Sections 3 to 6. Because 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], [3], [8], [23], [25], [31], [39], [40], [42].

C. An Illustrative Example

We now walk through how D* can be applied to generate a ranking of statements for fault localization. Consider the program in Fig. 3, which computes the sum of two integers. It has an

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

Statement	Program(P)	Coverage											
		t_1	t_2	t_3	t_4	t_5	t_6	t_7	t_8	t_9	t_{10}	t_{11}	t_{12}
s_1	read (a,b);	•	•	•	•	•	•	•	•	•	•	•	•
s_2	if (a < 10 && b < 10)	•	•	•	•	•	•	•	•	•	•	•	•
s_3	result = a − b; //Correct: a + b	•	•	•	•		•	•	•	•	•	•	•
s_4	if (result > 0)	•	•	•	•		•	•	•	•	•	•	•
s_5	print ("positive");	•					•	•			•	•	•
s_6	else if (result == 0)		•	•	•				•	•			
s_7	print ("zero");		•							•			
s_8	else print ("negative");			•	•				•				
s_9	else print ("invalid input");					•							
Execution Results (0=Successful / 1=Failed)		1	1	1	1	0	0	0	0	0	0	0	0

Fig. 3. Sample program with coverage and execution results for each test case.

TABLE I
COMPUTATION OF SUSPICIOUSNESS USING KULCZYNSKI, D^2, D^3

Statement	N_{CF}	N_{UF}	N_{CS}	Kulczynski (D^1)		D^2		D^3	
				Suspiciousness	Rank	Suspiciousness	Rank	Suspiciousness	Rank
s_1	4	0	8	0.50	5	2.00	4	8.00	4
s_2	4	0	8	0.50	5	2.00	4	8.00	4
s_3	4	0	7	0.57	3	2.29	2	9.14	1
s_4	4	0	7	0.57	3	2.29	2	9.14	1
s_5	1	3	5	0.13	8	0.13	8	0.13	8
s_6	3	1	2	1.00	1	3.00	1	9.00	3
s_7	1	3	1	0.25	7	0.25	7	0.25	7
s_8	2	2	1	0.67	2	1.33	6	2.67	6
s_9	0	4	1	0.00	9	0.00	9	0.00	9

error with regard to the sum computation in statement s_3, i.e., instead of adding the two integers, we accidentally subtract them. We also have a set of twelve test cases out of which eight execute successfully ($t_5, t_6, t_7, t_8, t_9, t_{10}, t_{11}$, and t_{12}), while the other four (t_1, t_2, t_3, and t_4) result in failure. The coverage information for each test case is also shown where a dot indicates that the corresponding statement is covered, and the absence of a dot indicates that the statement is not covered by that test case.

We begin by first collecting the required statistics (i.e., N_{CF}, N_{CS}, and N_{UF}) with respect to each statement, as discussed in Section 2.2. These statistics are shown in Table I.

The suspiciousness of each statement in Columns 5, 7, and 9 of Table I are computed using $N_{CF}/(N_{UF} + N_{CS})$ (Kulczynski), $(N_{CF})^2/(N_{UF} + N_{CS})(D^2)$, and $(N_{CF})^3/(N_{UF} + N_{CS})(D^3)$, respectively. For Kulczynski, the suspiciousness of s_3 (the faulty statement) is 0.57, tied with s_4, but less than s_6 (1.00) and s_8 (0.67). As a result, three statements (s_6, s_8, and s_3) in the best case, and four (s_6, s_8, s_4, and s_3) in the worst case have to be examined to locate the fault. If D^2 is used, s_3 is tied with s_4 in suspiciousness, but only less than s_6. Hence, in the best case two statements (s_6, and s_3), and in the worst case three statements (s_6, s_4, and s_3), need to be examined. When the value of * is increased to 3 (namely, using D^3), the suspiciousness of s_3 is still the same as s_4, but higher than the other statements. Hence, only one statement (s_3) needs to be examined in the best case, and two (s_4, and s_3) in the worst case. This example clearly demonstrates how D^* can be used to produce a ranking of program statements, and lead programmers to faults effectively. It also suggests that the effectiveness of D^* increases if a larger value of * is used.

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

III. CASE STUDIES

We first describe our experimental design in detail (Sections 3.1 to 3.4), and then move on to presenting the results of the case studies (Section 3.5).

TABLE II
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	23	211
	grep	12653	18	470
	make	20014	29	793
	Ant	75333	23	871
	space	9126	34	13585
	flex	13892	22	525
	sed	12062	20	360

A. Subject Programs

In our experiments, we use nine different sets of programs: the *Siemens suite*, the *Unix suite*, *gzip*, *grep*, *make*, *Ant*, *space*, *flex*, and *sed*. These sets result in 24 different subject programs. An individual description of each set is as follows, with a comprehensive summary in Table II.

The seven programs of the *Siemens* suite have been employed by many fault localization studies [8], [23], [25], [39], [40], [42]. All the correct and faulty versions of the programs and test cases were downloaded from [32]. The *Unix* suite consists of ten Unix utility programs, and because these programs have been so thoroughly used in industry and research alike, they can be a reliable, credible basis for our evaluations. The *Unix* suite has also been used in other studies such as [43]. The *space* program developed at the European Space Agency provides an interface that allows the user to describe the configuration of an array of antennas by using a high level language. The correct version, its faulty versions, and a test suite were obtained from [33].

Version 1.1.2 of the *gzip* program (which reduces the size of named files) was downloaded from [33]. Also found at [33] were versions 2.2 of the *grep* program (which searches for a pattern in a file), 3.76.1 of the *make* program (which manages building of executables and other products from source code), 1.1 of the *flex* program (which generates scanners that perform lexical pattern-matching on text), and 2.0 of the *sed* program (which parses textual input and applies user-specified transformation to it). Because the above programs are all written in the C language, we also decided to use the Java-based *Ant* program (version 1.6 beta) from the same website. Also packaged with each of these programs was a set of test cases and faulty versions.

For the *gzip*, *grep*, *make*, *flex*, *sed*, and *Ant* programs, further faulty versions were created using mutation-based fault injection, in addition to the ones that were downloaded. This creation was done to enlarge our data sets, and because mutation-based faults have been shown to simulate realistic faults well and provide reliable, trustworthy results [4], [11], [25], [28]. More discussion on this topic appears in Section 7.3. For *grep*, additional faults from [25] 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. In this way, we assure that each fault examined would be revealed by at least one test case.

TABLE III
SIMILARITY COEFFICIENT-BASED TECHNIQUES USED FOR COMPARISON

	Coefficient	Algebraic Form		Coefficient	Algebraic Form
1	Braun-Banquet	$\frac{N_{CF}}{max(N_{CF}+N_{CS}, N_{CF}+N_{UF})}$	17	Harmonic Mean[(1)]	$\frac{(N_{CF}\times N_{US}-N_{UF}\times N_{CS})((N_{CF}+N_{CS})\times(N_{US}+N_{UF})+(N_{CF}+N_{UF})\times(N_{CS}+N_{US}))}{(N_{CF}+N_{CS})\times(N_{US}+N_{UF})\times(N_{CF}+N_{UF})\times(N_{CS}+N_{US})}$
2	Dennis	$\frac{(N_{CF}\times N_{US})-(N_{CS}\times N_{UF})}{\sqrt{n\times(N_{CF}+N_{CS})\times(N_{CF}+N_{UF})}}$	18	Rogot2[(1)]	$\frac{1}{4}(\frac{N_{CF}}{N_{CF}+N_{CS}}+\frac{N_{CF}}{N_{CF}+N_{UF}}+\frac{N_{US}}{N_{US}+N_{CS}}+\frac{N_{US}}{N_{US}+N_{UF}})$
3	Mountford	$\frac{N_{CF}}{0.5\times((N_{CF}\times N_{CS})+(N_{CF}\times N_{UF}))+(N_{CS}\times N_{UF})}$	19	Simple Matching[(2)]	$\frac{N_{CF}+N_{US}}{N_{CF}+N_{CS}+N_{US}+N_{UF}}$
4	Fossum	$\frac{n\times(N_{CF}-0.5)^2}{(N_{CF}+N_{CS})\times(N_{CF}+N_{UF})}$	20	Rogers& Tanimoto[(2)]	$\frac{N_{CF}+N_{US}}{N_{CF}+N_{US}+2(N_{UF}+N_{CS})}$
5	Pearson	$\frac{n\times((N_{CF}\times N_{US})-(N_{CS}\times N_{UF}))^2}{N_C\times N_U\times N_S\times N_F}$	21	Hamming[(2)]	$N_{CF}+N_{US}$
6	Gower	$\frac{N_{CF}+N_{US}}{\sqrt{N_F\times N_C\times N_U\times N_S}}$	22	Hamann[(2)]	$\frac{N_{CF}+N_{US}-N_{UF}-N_{CS}}{N_{CF}+N_{UF}+N_{CS}+N_{US}}$
7	Michael	$\frac{4\times((N_{CF}\times N_{US})-(N_{CS}\times N_{UF}))}{(N_{CF}+N_{US})^2+(N_{CS}+N_{UF})^2}$	23	Sokal[(2)]	$\frac{2(N_{CF}+N_{US})}{2(N_{CF}+N_{US})+N_{UF}+N_{CS}}$
8	Pierce	$\frac{(N_{CF}\times N_{UF})+(N_{UF}\times N_{CS})}{(N_{CF}\times N_{UF})+(2\times(N_{UF}\times N_{US}))+(N_{CS}\times N_{US})}$	24	Scott[(3)]	$\frac{4(N_{CF}\times N_{US}-N_{UF}\times N_{CS})-(N_{UF}-N_{CS})^2}{(2N_{CF}+N_{UF}+N_{CS})(2N_{US}+N_{UF}+N_{CS})}$
9	Baroni-Urbani & Buser	$\frac{\sqrt{(N_{CF}\times N_{US})}+N_{CF}}{\sqrt{(N_{CF}\times N_{US})}+N_{CF}+N_{CS}+N_{UF}}$	25	Rogot1[(3)]	$\frac{1}{2}(\frac{N_{CF}}{2N_{CF}+N_{UF}+N_{CS}}+\frac{N_{US}}{2N_{US}+N_{UF}+N_{CS}})$
10	Tarwid	$\frac{(n\times N_{CF})-(N_F\times N_C)}{(n\times N_{CF})+(N_F\times N_C)}$	26	Kulczynski[(4)]	$\frac{N_{CF}}{N_{UF}+N_{CS}}$
11	Ample	$\left\lvert\frac{N_{CF}}{N_{CF}+N_{UF}}-\frac{N_{CS}}{N_{CS}+N_{US}}\right\rvert$	27	Anderberg[(4)]	$\frac{N_{CF}}{N_{CF}+2(N_{UF}+N_{CS})}$
12	Phi (Geometric Mean)	$\frac{N_{CF}\times N_{US}-N_{UF}\times N_{CS}}{\sqrt{(N_{CF}+N_{CS})\times(N_{CF}+N_{UF})\times(N_{CS}+N_{US})\times(N_{UF}+N_{US})}}$	28	Dice[(4)]	$\frac{2N_{CF}}{N_{CF}+N_{UF}+N_{CS}}$
13	Arithmetic Mean	$\frac{2(N_{CF}\times N_{US}-N_{UF}\times N_{CS})}{(N_{CF}+N_{CS})\times(N_{US}+N_{UF})+(N_{CF}+N_{UF})\times(N_{CS}+N_{US})}$	29	Goodman[(4)]	$\frac{2N_{CF}-N_{UF}-N_{CS}}{2N_{CF}+N_{UF}+N_{CS}}$
14	Cohen	$\frac{2(N_{CF}\times N_{US}-N_{UF}\times N_{CS})}{(N_{CF}+N_{CS})\times(N_{US}+N_{CS})+(N_{CF}+N_{UF})\times(N_{UF}+N_{US})}$	30	Jaccard[(4)]	$\frac{N_{CF}}{N_{CF}+N_{UF}+N_{CS}}$
15	Fleiss	$\frac{4(N_{CF}\times N_{US}-N_{UF}\times N_{CS})-(N_{UF}-N_{CS})^2}{(2N_{CF}+N_{UF}+N_{CS})+(2N_{US}+N_{UF}+N_{CS})}$	31	Sorensen-Dice[(4)]	$\frac{2N_{CF}}{2N_{CF}+N_{UF}+N_{CS}}$
16	Zoltar	$\frac{N_{CF}}{N_{CF}+N_{UF}+N_{CS}+\frac{10000\times N_{UF}\times N_{CS}}{N_{CF}}}$			

*Coefficients with the same superscript enclosed in parentheses are equivalent with each other. For example, Harmonic Mean and Rogot2 are equivalent as they can be simplified to the same formula.

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

B. Fault Localization Techniques Used in Comparisons

Because 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 30 other coefficients [20], [27] listed in Table III, along with their algebraic forms based on the notations described in the Notation section. These coefficients have been used in different studies such as [7], [37]. 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.

Some coefficients are proved to be *equivalent* from the fault localization point of view because they always produce the same suspiciousness rankings [10], [19], [27]. However, this result only implies the same statement has the same position in different rankings, but not necessarily the same suspiciousness. For example, Simple-Matching, Rogers & Tanimoto, Hamming, Hamann, and Sokal are equivalent to each other as they can all be simplified to $(N_{CF}+N_{US})/(N_{UF}+N_{CS})$; and Kulczynski, Anderberg, Dice, Goodman, Jaccard, and Sorensen-Dice are equivalent to $N_{CF}/(N_{UF}+N_{CS})$. Coefficients Scott and Rogot1 are also equivalent because Scott equals $4\times\text{Rogot1}-1$. Moreover, we prove that Harmonic Mean and Rogot2 are equivalent (as Rogot2 equals $(\text{Harmonic Mean}+2)/4$) even though this has not been previously shown in [10], [19], [27].

As per this section, D* has been compared to a total of 31 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 contemporary fault localization techniques (such as Crosstab [42], RBF [40], H3b and H3c [39], Tarantula [23], as well as Ochiai [1], [29] and Ochiai2 [27], [29]), and present the corresponding results in Section 5.

C. Evaluation Metrics and 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.

1) The EXAM Score: Renieris *et al.* [31] 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 [8], and is defined as $1 - |N|/|PDG|$, where $|N|$ is the number of all nodes examined, and $|PDG|$ is the number of all nodes in the program dependence graph. Instead of using such a graph, the Tarantula fault localization technique [23] directly uses the program's source code. To make their effectiveness computation comparable to those of the program dependence graph, Jones *et al.* [23] consider only executable statements to compute their score.

However, while the authors of [23] define their score to be the percentage of code that need not be examined to find a fault, we feel it is more straightforward to present the percentage of code that *has to be examined* to find the fault. This modified score is hereafter referred to as EXAM, and is defined as the percentage of statements that need to be examined until the first faulty statement is reached. Note that our main objective is to provide a good starting point for programmers to begin fixing a bug, rather than identifying all the code that would need to be corrected. From this perspective, even though a fault may span multiple, non-contiguous statements, the fault localization process is halted once the first statement corresponding to the bug is found. Many other fault localization studies observe the same philosophy.

A similar such modification is made by the authors of [25]. They define their effectiveness (T-score) as $T = (|V_{examined}|/|V|) \times 100\%$, 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.

In short, 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).

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 ith 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) Wilcoxon Signed-Rank Test: Additionally, 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) [30]. Because we aim to show that D* is more effective than other fault localization techniques, we evaluate the one-tailed alternative hypothesis that the other techniques require the examination of an equal or greater number of statements than D*. The null hypothesis in this case specifies that the other techniques require the examination of a number of statements that is less than required by D*). Stated differently,

H_o: Number of statements examined by other techniques $<$ Number of statements examined by D*

Thus, 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.

D. Data Collection

For the *Siemens* suite and the *Unix* suite, all executions were on a PC with a 2.13 GHz Intel Core 2 Duo CPU and 8 GB physical memory. The operating system was SunOS 5.10 (Solaris 10), and the compiler used was gcc 3.4.3. For *grep*, *gzip*, *make*, *space*, *flex*, and *sed*, the executions were on a Sun-Fire-280R machine with SunOS 5.10 as the operating system, and gcc 3.4.4 as the compiler. Executions for the *Ant* program were on the same machine as for the *Siemens* suite and the *Unix* suite, and the Java compiler was version 1.5.0_06.

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 a revised version of χ Suds [45]. This version not only records which statements of the source code are executed by a test case, but also how many times each statement is executed. 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 [9] was used to instrument the code and collect coverage data.

E. 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 condition 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. In all our experiments, we assume that for the best effectiveness the faulty statement is at the top of the list of statements with same suspiciousness, and for the worst effectiveness it is at the end of the list. During our evaluation of D*, data is presented for these two levels of effectiveness, across all the evaluation metrics (criteria).

Currently, each faulty version under consideration has exactly one bug, potentially involving multiple statements at different locations in the code or different functions. Many other studies, such as [8], [23]–[25], take a similar approach. However, our proposed technique is also capable of handling programs with multiple bugs, which is further discussed in Section 6. Also, this

TABLE IV
TOTAL NUMBER OF STATEMENTS EXAMINED TO LOCATE FAULTS FOR EACH OF THE SUBJECT PROGRAMS (BEST & WORST CASES)

	Best Case									Worst Case								
	Unix	*Siemens*	*grep*	*gzip*	*make*	*Ant*	*space*	*flex*	*sed*	*Unix*	*Siemens*	*grep*	*gzip*	*make*	*Ant*	*space*	*flex*	*sed*
D^2	1805	1754	2596	1215	7689	672	1300	810	1787	5226	2650	4053	2681	13130	1184	2375	1493	3937
Braun-Banquet	2767	2438	3507	1353	8667	2196	2630	1121	2259	6187	3296	4963	2804	14393	2698	3705	1850	4391
Dennis	2934	2206	4869	1955	11544	1974	2253	1651	2316	6504	3074	8030	3406	16757	2476	3328	3499	4448
Mountford	2183	1974	2991	1297	8471	3298	1801	1146	2092	5644	2832	4453	2762	13828	3818	2876	1864	4224
Fossum	2468	2230	13817	4542	16966	150415	4489	1816	5358	5843	3126	18781	6633	22179	150917	5564	2530	7490
Pearson	3581	3279	6003	1445	14515	1188	5766	5172	2837	7221	4247	9601	2896	19855	1690	6942	7068	4969
Gower	8630	6586	40877	22754	122073	967307	59315	48982	36375	12027	7434	42434	24205	127286	967809	63494	49696	38507
Michael	3713	1993	4400	2312	11694	4502	5425	1806	2451	7283	2864	7597	3763	16907	5004	6557	3654	4583
Pierce	11782	8072	15828	19222	28221	322033	57492	39485	24903	23387	15299	57131	38404	154221	1018725	106168	75918	63458
Baroni-Urbani & Buser	3189	3547	4080	1423	9027	4693	5037	986	3356	6605	4404	5536	2874	14240	5195	6112	1700	5488
Tarwid	3399	2453	5164	3105	13415	5964	3608	6527	4061	7883	3321	8611	4605	19467	9935	4826	8817	6655
Ample	3757	3336	7061	2032	16895	1801	6226	5315	3757	7378	4232	10582	3483	22141	3367	7303	7163	5889
Phi (Geometric Mean)	2524	2005	3486	1309	9409	1076	1790	1336	2068	6094	2873	6647	2760	14622	1578	2865	3184	4200
Arithmetic Mean	2615	2049	3683	1316	9621	1399	2031	1347	2014	6185	2917	6844	2767	14834	1901	3106	3195	4146
Cohen	2916	2419	3960	1338	9730	1837	2553	1383	2249	6486	3287	7121	2789	14943	2339	3628	3231	4381
Fleiss	4710	5631	17829	7377	33852	89263	14639	3741	8441	8083	6487	19285	8828	39065	89765	15714	4455	10573
Zoltar	3449	1737	33634	12766	93284	709633	23497	20669	25633	6881	2635	35090	14217	98497	710135	27658	21383	27765
Harmonic Mean[1]	2675	2274	4021	1297	7745	1283	1770	2888	2053	6324	3142	7182	2748	12958	1785	2845	4736	4185
Rogot2[1]	2675	2274	4021	1297	7745	1283	1770	2888	2053	6324	3142	7182	2748	12958	1785	2845	4736	4185
Simple Matching[2]	5545	6335	20952	9082	35461	250414	24352	6721	9895	8977	7187	22475	10637	41493	253631	25450	7466	13164
Rogers & Tanimoto[2]	5545	6335	20952	9082	35461	250414	24352	6721	9895	8977	7187	22475	10637	41493	253631	25450	7466	13164
Hamming[2]	5545	6335	20952	9082	35461	250414	24352	6721	9895	8977	7187	22475	10637	41493	253631	25450	7466	13164
Hamann[2]	5545	6335	20952	9082	35461	250414	24352	6721	9895	8977	7187	22475	10637	41493	253631	25450	7466	13164
Sokal[2]	5545	6335	20952	9082	35461	250414	24352	6721	9895	8977	7187	22475	10637	41493	253631	25450	7466	13164
Scott[3]	4727	5564	17812	6964	24992	90160	14568	3805	8429	8100	6420	19268	8415	30205	90662	15643	4519	10561
Rogot1[3]	4727	5564	17812	6964	24992	90160	14568	3805	8429	8100	6420	19268	8415	30205	90662	15643	4519	10561
Kulcynzki[4]	2361	2327	2971	1268	7909	1565	2107	856	2083	5777	3186	4427	2719	13283	2067	3182	1570	4215
Anderberg[4]	2361	2327	2971	1268	7909	1565	2107	856	2083	5777	3186	4427	2719	13283	2067	3182	1570	4215
Dice[4]	2361	2327	2971	1268	7909	1565	2107	856	2083	5777	3186	4427	2719	13283	2067	3182	1570	4215
Goodman[4]	2361	2327	2971	1268	7909	1565	2107	856	2083	5777	3186	4427	2719	13283	2067	3182	1570	4215
Jaccard[4]	2361	2327	2971	1268	7909	1565	2107	856	2083	5777	3186	4427	2719	13283	2067	3182	1570	4215
Sorensen-Dice[4]	2361	2327	2971	1268	7909	1565	2107	856	2083	5777	3186	4427	2719	13283	2067	3182	1570	4215

∗Coefficients with the same superscript enclosed in parenthesis are equivalent with each other.

section focuses on D^* where * equals 2, with the effectiveness of higher values to be discussed in Sections 4 and 5.

Table IV presents the cumulative number of statements that need to be examined by each fault localization technique across each of the subject programs under study, for both best and worst cases. For example, we find that, for the *Ant* program, D^2 can locate faults in all faulty versions by requiring the examination of no more than 672 statements in the best case, and 1184 in the worst. Again, with respect to the *Ant* program, we find that the second best technique is Phi (also known as the Geometric Mean), which can locate all the faults by requiring the examination of no more than 1076 statements in the best case, and 1578 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 IV by the corresponding number of faulty versions (for that program) in Table II. For example, on *Ant*, in the best case, D^2 requires the examination of about 30 (672/23) statements (only 0.0398% (30/75333) of the code) per faulty version, and about 52 (1184/23) statements (0.069% (52/75333) of the code) in the worst.

From Table IV, 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 except for (i) the *Siemens* suite where Zoltar is the most effective and D^2 is the second, and (ii) the worst case of the *make* program where Harmonic Mean and Rogot2 are the most effective and D^2 is the second; however, for the best case of the *make* program it is D^2 as the most effective, and Harmonic Mean and Rogot2 are second.

We re-emphasize that D^2 is the most effective technique on *most* of the subject programs under study (more precisely, 15 of the 18 scenarios except for the best and worst cases of *Siemens*, and the worst case of *make*, where D^2 is the second most effective). 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, Kulczynski (and its equivalents), or Harmonic Mean (and its equivalent) seem to be second best, this is not the case throughout all subject programs. More precisely, Mountford is only the second best for 2 of the 18 aforementioned scenarios, Kulczynski for 6 of 18, and Harmonic Mean for 3 of 18 (with one scenario where it is the best). Furthermore, there are scenarios where the second best technique is none of these three. For example, consider that on the *sed* and *Ant* programs (irrespective of best or worst case), the Arithmetic Mean and Phi are the second most effective, respectively.

Another significant point worth noting is that often the worst case of D^2 is better than or at least comparable to the best case of the competing techniques. For example, with respect to the *Ant* program, the worst case of $D^2(1184)$ is still better than the best case of Mountford (3298), Harmonic Mean and its equivalent (1283), and Kulczynski and its equivalents (1565).

The above observations justify 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 now present the evaluation of D^2 with respect to the EXAM score. To give a flavor of the results without burdening the reader with too much data, we only show figures for three

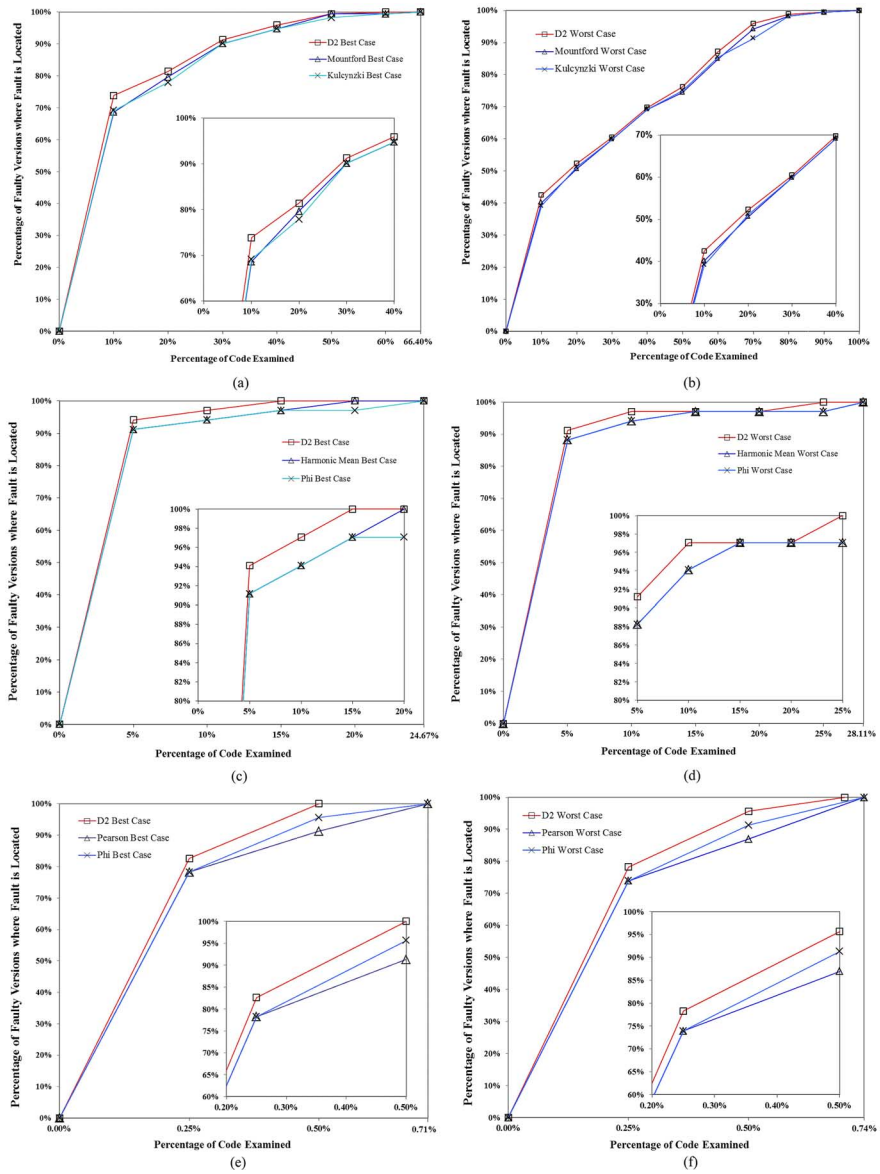

Fig. 4. EXAM score-based comparison between D^2 and other similarity coefficient-based techniques.(a) Best case on *Unix*, (b) Worst case on *Unix*, (c) Best case on *space*, (d) Worst case on *space*, (e) Best case on *Ant*, (f) Worst case on *Ant*.

programs: *Unix*, *space*, and *Ant*. Figures for the other 6 sets of programs (*Siemens*, *grep*, *gzip*, *make*, *flex*, and *sed*) were generated but not included here.[5] However, the conclusions drawn with respect to the first three programs are also applicable to the remaining six.

In each figure, the x-axis represents the percentage of code (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 parts (a) and (b) of Fig. 4, we find that on the *Unix* suite, by examining less than 10% of the code, D^2 can locate faults for 73.83% of the faulty versions in the best case, and 42.44% in the worst. In contrast, by examining less than 10% of the code, Mountford (the second best technique) can only locate 68.60% of the faults in the best case, and 40.12% in the worst. The percentage for Kulczynski (the third best) is 69.18% (best case), and 39.08% (worst case). The curves in parts (c) and (d) are the three techniques with the top performance for the *space* program, and those in parts (e) and (f) are the best three techniques for the *Ant* program. They clearly indicate the superiority of D^2 over the other techniques when evaluated using the EXAM score. The reason why only three curves are displayed in each part of Fig. 4 is purely in the interests of clarity and readability. Note that each technique evaluated would have two curves corresponding to the best and worst cases of that technique, respectively. Including each of the 32 techniques (31 competing techniques plus D^2) would have meant 32 different curves in each figure, which makes it very difficult, if not entirely impossible, to decipher any useful information from the figures. Nevertheless, we emphasize that, while the curves with respect to other similarity coefficient-based techniques in Table III have not been presented in Fig. 4, the corresponding curves were nevertheless generated;[5] and with respect

[5]Readers interested in obtaining the complete set of figures and curves may contact the authors.

TABLE V
CONFIDENCE WITH WHICH IT CAN BE CLAIMED THAT D* IS MORE EFFECTIVE THAN OTHER TECHNIQUES (BEST AND WORST CASES)

	Unix		*Siemens*		*grep*		*gzip*		*make*		*Ant*		*space*		*flex*		*sed*	
	Best	Worst	Best	Worst	Best	Worst	Best	Worst	Best	Worst	Best	Worst	Best	Worst	Best	Worst	Best	Worst
Braun-Banquet	99.99%	99.99%	99.99%	99.99%	99.90%	99.85%	99.81%	98.05%	99.99%	99.55%	99.81%	99.22%	99.98%	99.99%	99.02%	99.93%	99.95%	99.27%
Dennis	99.99%	99.99%	99.99%	99.99%	99.61%	99.61%	99.99%	99.90%	99.99%	99.99%	99.81%	99.22%	98.44%	98.44%	99.90%	99.99%	99.90%	98.54%
Mountford	99.99%	99.99%	99.99%	99.99%	99.22%	99.81%	93.75%	87.50%	99.81%	97.63%	99.90%	99.81%	96.88%	96.88%	96.88%	99.81%	99.90%	98.93%
Fossum	99.99%	99.99%	99.99%	99.99%	99.81%	99.95%	99.22%	98.83%	99.61%	91.99%	99.22%	96.88%	99.61%	99.61%	93.75%	99.22%	98.44%	92.19%
Pearson	99.99%	99.99%	99.99%	99.99%	99.90%	99.99%	99.22%	94.53%	99.99%	99.99%	99.22%	96.88%	99.81%	99.90%	99.22%	99.90%	99.61%	98.05%
Gower	99.99%	99.99%	99.99%	99.99%	99.99%	99.99%	99.99%	99.99%	99.99%	99.99%	99.99%	99.99%	99.99%	99.99%	99.99%	99.99%	99.99%	99.99%
Michael	99.99%	99.99%	99.69%	99.54%	99.98%	99.98%	99.99%	99.99%	99.99%	99.99%	99.97%	99.97%	99.99%	99.99%	99.02%	99.02%	99.97%	99.96%
Pierce	99.99%	99.99%	99.99%	99.99%	99.99%	99.99%	99.99%	99.99%	99.99%	99.99%	99.99%	99.99%	99.99%	99.99%	99.99%	99.99%	99.99%	99.99%
Baroni-Urbani & Buser	99.99%	99.99%	99.99%	99.99%	99.90%	99.85%	99.81%	98.63%	99.99%	99.63%	99.81%	98.83%	99.81%	99.81%	96.88%	99.61%	99.81%	98.44%
Tarwid	99.99%	99.99%	99.99%	99.99%	99.98%	99.99%	99.99%	99.99%	99.99%	99.99%	99.99%	99.99%	99.99%	99.99%	99.99%	99.99%	99.99%	99.99%
Ample	99.99%	99.99%	99.99%	99.99%	99.90%	99.99%	99.98%	99.54%	99.99%	99.99%	95.70%	98.44%	99.51%	99.46%	99.81%	99.98%	99.41%	97.46%
Phi (Geometric Mean)	99.99%	99.99%	99.99%	99.99%	99.61%	99.61%	96.88%	90.63%	99.99%	99.93%	99.22%	96.88%	87.50%	87.50%	96.88%	99.61%	99.22%	96.09%
Arithmetic Mean	99.99%	99.99%	99.99%	99.99%	99.61%	99.61%	98.44%	92.19%	99.99%	99.93%	99.22%	96.88%	96.88%	96.88%	96.88%	99.61%	99.61%	97.27%
Cohen	99.99%	99.99%	99.99%	99.99%	99.81%	99.81%	99.81%	97.27%	99.99%	99.93%	99.61%	98.44%	99.22%	99.22%	99.22%	99.90%	99.81%	97.85%
Fleiss	99.99%	99.99%	99.99%	99.99%	99.90%	99.90%	99.98%	99.88%	99.99%	99.99%	99.99%	99.99%	99.51%	99.51%	99.61%	99.95%	99.41%	96.48%
Zoltar	99.99%	99.99%	93.14%	93.19%	99.99%	99.99%	99.99%	99.99%	99.99%	99.99%	99.99%	99.99%	99.99%	99.99%	99.99%	99.99%	99.99%	99.99%
Harmonic Mean[1]	99.99%	99.99%	99.99%	99.99%	99.02%	98.93%	93.75%	87.50%	82.45%	80.12%	94.92%	89.06%	98.13%	98.13%	96.88%	99.61%	98.44%	92.19%
Rogot2[1]	99.99%	99.99%	99.99%	99.99%	99.02%	98.93%	93.75%	87.50%	82.45%	80.12%	94.92%	89.06%	98.13%	98.13%	96.88%	99.61%	98.44%	92.19%
Simple Matching[2]	99.99%	99.99%	99.99%	99.99%	99.98%	99.99%	99.81%	99.71%	99.98%	99.99%	99.90%	99.81%	99.99%	99.99%	98.44%	99.95%	99.90%	99.22%
Rogers & Tanimoto[2]	99.99%	99.99%	99.99%	99.99%	99.98%	99.99%	99.81%	99.71%	99.98%	99.99%	99.90%	99.81%	99.99%	99.99%	98.44%	99.95%	99.90%	99.22%
Hamming[2]	99.99%	99.99%	99.99%	99.99%	99.98%	99.99%	99.81%	99.71%	99.98%	99.99%	99.90%	99.81%	99.99%	99.99%	98.44%	99.95%	99.90%	99.22%
Hamann[2]	99.99%	99.99%	99.99%	99.99%	99.98%	99.99%	99.81%	99.71%	99.98%	99.99%	99.90%	99.81%	99.99%	99.99%	98.44%	99.95%	99.90%	99.22%
Sokal[2]	99.99%	99.99%	99.99%	99.99%	99.98%	99.99%	99.81%	99.71%	99.98%	99.99%	99.90%	99.81%	99.99%	99.99%	98.44%	99.95%	99.90%	99.22%
Scott[3]	99.99%	99.99%	99.99%	99.99%	99.90%	99.90%	99.81%	99.41%	99.99%	99.99%	99.61%	98.44%	99.90%	99.90%	99.22%	99.90%	99.90%	98.54%
Rogot1[3]	99.99%	99.99%	99.99%	99.99%	99.90%	99.90%	99.81%	99.41%	99.99%	99.99%	99.61%	98.44%	99.90%	99.90%	99.22%	99.90%	99.90%	98.54%
Kulcynzki[4]	99.99%	99.99%	99.99%	99.99%	99.22%	99.22%	96.88%	94.38%	96.88%	91.25%	99.61%	97.66%	99.22%	99.22%	96.88%	99.61%	99.61%	96.09%
Anderberg[4]	99.99%	99.99%	99.99%	99.99%	99.22%	99.22%	96.88%	94.38%	96.88%	91.25%	99.61%	97.66%	99.22%	99.22%	96.88%	99.61%	99.61%	96.09%
Dice[4]	99.99%	99.99%	99.99%	99.99%	99.22%	99.22%	96.88%	94.38%	96.88%	91.25%	99.61%	97.66%	99.22%	99.22%	96.88%	99.61%	99.61%	96.09%
Goodman[4]	99.99%	99.99%	99.99%	99.99%	99.22%	99.22%	96.88%	94.38%	96.88%	91.25%	99.61%	97.66%	99.22%	99.22%	96.88%	99.61%	99.61%	96.09%
Jaccard[4]	99.99%	99.99%	99.99%	99.99%	99.22%	99.22%	96.88%	94.38%	96.88%	91.25%	99.61%	97.66%	99.22%	99.22%	96.88%	99.61%	99.61%	96.09%
Sorensen-Dice[4]	99.99%	99.99%	99.99%	99.99%	99.22%	99.22%	96.88%	94.38%	96.88%	91.25%	99.61%	97.66%	99.22%	99.22%	96.88%	99.61%	99.61%	96.09%

∗Coefficients with the same superscript enclosed in parentheses are equivalent with each other.

to the EXAM score, D^2 performs better than those techniques. This result is consistent with our observations from Fig. 4 that D^2 is the most effective technique except for the best and worst cases of the *Siemens* suite, and the worst case of the *make* program, where it is the second most effective.

With respect to our third metric of evaluation, Table V 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 a given technique to find faults, thereby making it more effective) can be accepted with respect to a selected program and the corresponding best or worst case.

To take an example, it can be said with 99.99% confidence that D^2 is more effective than Braun-Banquet on the *Siemens* suite for both the best and the worst cases. Similar observations can also be made for most of the scenarios in Table V with 99% confidence except for a few, such as D^2 being more effective than Zoltar with 93.14% confidence for the best case of *Siemens*, and with 98.54% confidence being better than Dennis for the worst case of the *sed* program. Of the 558 scenarios (31 techniques each with best and worst cases for 9 sets of programs), only 9 have confidence less than 90% with the lowest being 80.12% for Harmonic Mean and Rogot2 with respect to the worst case of the *make* program.

Let us now modify our alternative hypothesis to consider equalities. Thus, we 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. Based on the data[6] with respect to this modified alternative hypothesis, we observe an increase to 100% of many confidence values in Table V with respect to the original alternative. Note also each confidence value can only be raised (but not lowered) with respect to the new alternative hypothesis. Of the 558 scenarios, only 4 confidence values are less than 90% with 83.24% being the lowest for Harmonic Mean and Rogot2 with respect to the best case of the *make* program, which is in no way a low confidence. Moreover, if we also consider the fact that D^2 is in general better than these two techniques in terms of the EXAM score (for example, see the curves in Parts (c) and (d) of Fig. 4) and the total number of statements that need to be examined to find all the faults (refer to Table IV), we can comfortably conclude that D^2 is better than Harmonic Mean and Rogot2 as a fault localization technique.

The same conclusion also applies to other competing techniques. In summary, our case studies clearly show that D^2 is more effective than the similarity coefficient-based techniques in Table III at fault localization.

IV. EFFECTIVENESS OF D* WITH DIFFERENT VALUES FOR THE *

In this section, we investigate the fault localization effectiveness (in terms of the total number of statements examined) of

[6]We decide not to show such data as it is very similar to that in Table V except that many confidence values are now higher than before. Interested readers may contact the authors for the complete set of data.

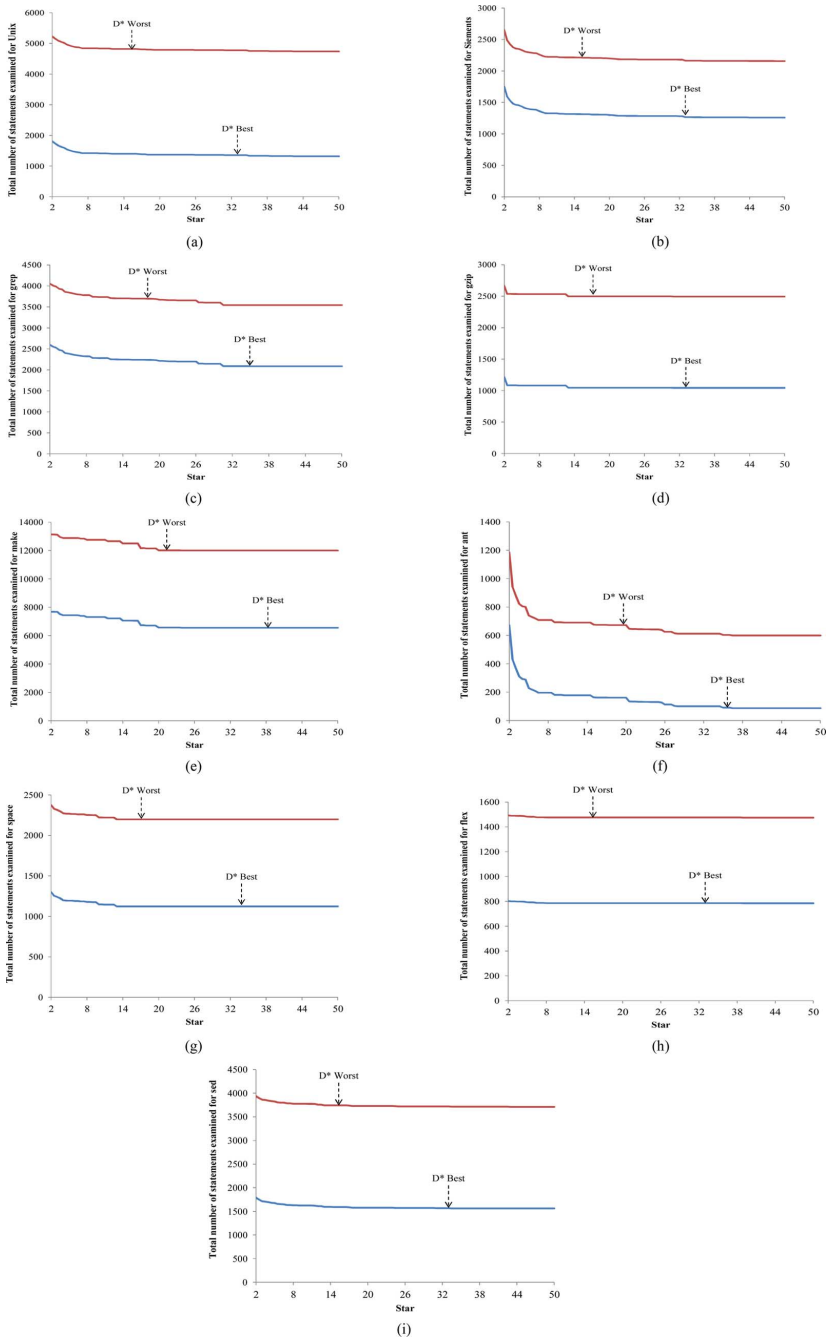

Fig. 5. Fault localization effectiveness of D* with respect to different values of *. (a) the *Unix* suite, (b) the *Siemens* suite, (c) the *gzip* program, (d) the *grep* program, (e) the *make* program, (f) the *Ant* program, (g) the *space* program, (h) the *flex* program. (i) the *sed* program.

D^* with respect to different values of $*$ followed by a detailed discussion of our observations.

A. Empirical Evaluation

We first empirically evaluate the impact different values of * have on the effectiveness of D^*. The range used is from 2 to 50 with an increment of 0.5 (namely, $* = 2.0, 2.5, 3.0, \ldots, 50.0$). From Fig. 5, we observe that the total number of statements examined to locate all the bugs in each set of programs declines (more precisely, the number of statements examined decreases in most cases while occasionally staying unchanged) as the value of the * increases, and after that the number remains the same.

For example, with respect to the *Ant* program, 672 statements need to be examined for D^2 in the best case, while the number decreases to 368 for D^3, and further down to 209 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.

B. Discussion of the Effectiveness of D^*

We now provide a more detailed discussion of how the effectiveness of D^* varies along with *. In particular, we address two points:

- Why does the effectiveness of D^* increase or remain the same as the value of * increases until it reaches a critical value?
- Why does the effectiveness of D^* eventually level off?

For discussion purposes, let us assume the program to be debugged has exactly one bug.[7] Assume also all the faulty statements are executed together. Let statement ω be the first faulty statement examined from the top of a suspiciousness ranking.

With respect to D^*, when * is equal to 1, it is the *Kulczynski* coefficient (i.e., $\mathrm{D}^1 = N_{CF}/(N_{UF} + N_{CS})$). The suspiciousness of ω can be simplified as $\mathrm{susp}(\omega) = N_F/N_{CS}(\omega)$ because $N_{UF}(\omega) = 0$ and $N_{CF}(\omega) = N_F$, and the suspiciousness of a non-faulty statement τ can be computed as $N_{CF}(\tau)/(N_{UF}(\tau) + N_{CS}(\tau)) = N_{CF}(\tau)/(N_F - N_{CF}(\tau) + N_{CS}(\tau))$. Similarly, with respect to D^2 (when $* = 2$), the suspiciousness of ω, and τ can be computed respectively as $(N_F)^2/N_{CS}(\omega)$, and $(N_{CF}(\tau))^2/(N_F - N_{CF}(\tau) + N_{CS}(\tau))$. Let $\mathcal{GREATER}, \mathcal{SAME}$, and $\mathcal{LESS}$ be the sets containing all the statements whose suspiciousness values are greater than, the same as, and less than that of ω. An additional superscript is included to emphasize the underlying fault localization technique, if necessary. For example, $\mathcal{GREATER}^{(1)}$ is the set for D^1, and $\mathcal{GREATER}^{(2)}$ is for D^2.

Because fault localization stops as soon as ω is reached, as per the discussion in the beginning of Section 3.5, in the best case we examine all the statements in $\mathcal{GREATER}$ and ω itself, whereas in the worst case we have to examine not only all the statements in $\mathcal{GREATER}$ but also every statement in $\mathcal{SAME}$. The authors of [47] have suggested a way to compare the effectiveness of two fault localization techniques using their $\mathcal{GREATER}$ and $\mathcal{LESS}$ sets. A potential problem of this approach is explained in the Appendix. In our case, if we can show that $\mathcal{GREATER}^{(2)} \subseteq \mathcal{GREATER}^{(1)}$, and $\mathcal{LESS}^{(2)} \supseteq \mathcal{LESS}^{(1)}$, then D^2 is more effective in fault localization than D^1.

To show $\mathcal{GREATER}^{(2)} \subseteq \mathcal{GREATER}^{(1)}$, we need to demonstrate that if a statement τ is in $\mathcal{GREATER}^{(2)}$, then τ must also be in $\mathcal{GREATER}^{(1)}$, but not vice versa. Stated differently, we need to show that if $(N_{CF}(\tau))^2/(N_F - N_{CF}(\tau) + N_{CS}(\tau)) > (N_F)^2/N_{CS}(\omega)$, then $N_{CF}(\tau)/(N_F - N_{CF}(\tau) + N_{CS}(\tau)) > N_F/N_{CS}(\omega)$, but not the other way around. We start by first assuming $(N_{CF}(\tau))^2/(N_F - N_{CF}(\tau) + N_{CS}(\tau)) > (N_F)^2/N_{CS}(\omega)$ is true. This assumption implies $N_{CF}(\tau) \times (N_{CF}(\tau)/(N_F - N_{CF}(\tau) + N_{CS}(\tau))) > N_F \times (N_F/N_{CS}(\omega))$. Because $N_{CF}(\tau) \leq N_F$, we conclude that $N_{CF}(\tau)/(N_F - N_{CF}(\tau) + N_{CS}(\tau)) > N_F/N_{CS}(\omega)$. Second, we examine the opposite direction by assuming $N_{CF}(\tau)/(N_F - N_{CF}(\tau) + N_{CS}(\tau)) > N_F/N_{CS}(\omega)$ is true. However, because $N_{CF}(\tau) \leq N_F$, the inequality $(N_{CF}(\tau))^2/(N_F - N_{CF}(\tau) + N_{CS}(\tau)) > (N_F)^2/N_{CS}(\omega)$ may or may not hold. If it does, together with the first part, we have $\mathcal{GREATER}^{(2)} = \mathcal{GREATER}^{(1)}$; otherwise, we have $\mathcal{GREATER}^{(2)} \subset \mathcal{GREATER}^{(1)}$. Thus, overall we have $\mathcal{GREATER}^{(2)} \subseteq \mathcal{GREATER}^{(1)}$. The procedure to show $\mathcal{LESS}^{(2)} \supseteq \mathcal{LESS}^{(1)}$ can be accomplished in a similar fashion. In summary, this result explains why D^2 is more effective than or as effective as D^1. Similarly, we can show that D^3 is more effective than or as effective as D^2, etc.

[7] A discussion of programs with multiple bugs appears in Section 6.

We now discuss why the effectiveness of D^* levels off as the value of * continues to increase. The suspiciousness of ω and τ with respect to D^* can be computed as $(N_F)^*/N_{CS}(\omega)$ and $(N_{CF}(\tau))^*/(N_F - N_{CF}(\tau) + N_{CS}(\tau))$. Let $\mathcal{GS}^{(*)}$ be the union of $\mathcal{GREATER}^{(*)}$ and $\mathcal{SAME}^{(*)}$. Hence, if a statement τ is in $\mathcal{GS}^{(*)}$, its suspiciousness must satisfy the inequality

$$\frac{(N_{CF}(\tau))^*}{N_F - N_{CF}(\tau) + N_{CS}(\tau)} \geq \frac{(N_F)^*}{N_{CS}(\omega)} \tag{1}$$

Note that $N_F \geq 1$ because, per Section 3.1, if a fault cannot be detected by at least one test case, it is excluded. The two denominators can be viewed as positive constants for a given τ. Hence, $N_{CF}(\tau)$ cannot be zero. Otherwise, the inequality (1) cannot be satisfied. That is, $N_{CF}(\tau) \geq 1$. Moreover, we also have $N_{CF}(\tau) \leq N_F$. With this understanding, we also ask a related question: does the set $\mathcal{GS}^{(*)}$ become smaller as * increases?

Case I: $N_{CF}(\tau) = N_F$. For statements in this category, their membership in $\mathcal{GS}^{(*)}$ is not affected by the value of *. If they are in the set, they remain there. If they are not, they will not be included because of the variation of *. Hence, * has no impact on the size of $\mathcal{GS}^{(*)}$. The case condition also implies the effectiveness of D^* remains the same as * increases.

Case II: $N_{CF}(\tau) < N_F$. For such statements, we also have $N_F > 1$ and $(N_{CF}(\tau))^* < (N_F)^*$. That is, N_F will grow faster than $N_{CF}(\tau)$ as * increases. As a result, it is possible that we can find a statement $\tau \in \mathcal{GS}^{(*)}$ for smaller values of *, but this is not the case for larger values of *. More specifically, there exists τ such that $\tau \in \mathcal{GS}^{(\gamma)}$ but $\notin \mathcal{GS}^{(\xi)}$, where $\gamma < \xi$. This condition makes $\mathcal{GS}^{(\gamma)} \supset \mathcal{GS}^{(\xi)}$. Hence, under this condition, the size of $\mathcal{GS}^{(*)}$ becomes smaller, and the effectiveness of D^* increases as * increases.

When the value of * keeps increasing, eventually it reaches a critical value ψ_τ such that

$$\frac{(N_{CF}(\tau))^{\psi_\tau}}{N_F - N_{CF}(\tau) + N_{CS}(\tau)} = \frac{(N_F)^{\psi_\tau}}{N_{CS}(\omega)}$$

With respect to a statement τ, if $N_{CF}(\tau)/(N_F - N_{CF}(\tau) + N_{CS}(\tau)) \geq N_F/N_{CS}(\omega)$ (i.e., $\tau \in \mathcal{GS}^{(1)}$), then τ will remain in $\mathcal{GS}^{(*)}$ as long as $^* \leq \psi_\tau$. Once * exceeds ψ_τ, the statement τ will be removed from $\mathcal{GS}^{(*)}$, and cannot be included again. On the other hand, if τ is not in $\mathcal{GS}^{(1)}$, then it will not be in any of the $\mathcal{GS}^{(*)}$ for $^* \geq 1$. Thus, for each non-faulty statement τ, there is a corresponding critical value ψ_τ such that the effectiveness of D* may be improved *once* due to the removal of τ from $\mathcal{GS}^{(*)}$ when * goes beyond ψ_τ.

Because a program P has many non-faulty statements (i.e., many τ), the critical value ψ_P for the entire program is the maximum of all the ψ_τ. This explains why the effectiveness of D^* levels off after the value of * surpasses ψ_P, because every statement which can be removed from $\mathcal{GS}^{(*)}$ by increasing the value of * has already been removed. The remaining statements in $\mathcal{GS}^{(*)}$ will stay no matter how large * grows.

V. 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 re-

sults indicate that D^2 (where * has a value of 2) is more effective than other competing techniques. However, additional 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. Because it is impossible to compare D^* to all the techniques, we select a representative set, each of which is well-known and has been demonstrated to be effective, and compare D^* to each of them: Tarantula [23], Ochiai [1], [29], Ochiai2 [27], [29], Crosstab [42], RBF [40], and Heuristics H3b and H3c [39].

The Tarantula fault localization technique [23] assigns a suspiciousness value to each statement as per the formula $X/(X + Y)$, where $X = (N_{CF}/N_F)$ and $Y = (N_{CS}/N_S)$ based on the notation used in this paper. 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 [23]. In [1], Abreu *et al.* evaluate the use of the Ochiai coefficient [29] in the context of software fault localization, which assigns a suspiciousness value to each statement using the formula

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

The Ochiai2 coefficient, proposed in [29] and used as a fault localization technique in [27], is defined as the equation shown at the bottom of the page. Wong *et al.* presented a crosstab (cross tabulation) based statistical technique for fault localization [42]. A unique aspect of this technique is that it is based on a well-defined statistical analysis, which has been used to study the relationship between two or more categorical variables [13], [15], [17]. In this case, the two variables are *test execution result* (*success* or *failure*), and *the coverage of a given statement*. The null hypothesis is that they are statistically independent. However, instead of the so-called statistical dependence-independence relationship, we are more interested in the degree of association between the execution result and the coverage of each statement. More precisely, a crosstab is constructed for each statement with two column-wise categorical variables (*covered* and *not covered*) and two row-wise categorical variables (*successful* and *failed*) to help analyze whether there exists a high (or low) degree of association between the coverage of a statement and the failed (or successful) execution result. In addition, a statistic is computed based on each table to determine the suspiciousness of the corresponding statement.

Wong *et al.* also proposed a fault localization technique using a modified RBF (radial basis function) neural network [40]. Such a network consists of a three-layer feed-forward structure: one input layer, one output layer, and one hidden layer where each neuron uses a Gaussian basis function [18] as the activation function with the distance computed using weighted bit comparison-based dissimilarity. The network is trained to learn the relationship between the statement coverage of a test case and its corresponding success or failure. Once the training is complete, a set of virtual test cases (each covering a single statement) is provided as input, and the output for each virtual test case is considered to be the suspiciousness of the covered statement.

In addition, Wong *et al.* examined how each additional failed (or successful) test case can help locate program faults [39]. 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 $[(1.0) \times n_{\mathrm{F},1} + (0.1) \times n_{\mathrm{F},2} + (0.01) \times n_{\mathrm{F},3}] - [(1.0) \times n_{\mathrm{s},1} + (0.1) \times n_{\mathrm{s},2} + \alpha \times \chi_{\mathrm{F/S}} \times n_{\mathrm{s},3}]$, where $n_{\mathrm{F},i}$ and $n_{\mathrm{S},i}$ are the number of failed and successful tests in the ith group, and $\chi_{\mathrm{F/S}}$ 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 $\alpha = 0.0001$, which are different from the less effective heuristics with the same names in their earlier study [44]. The improvement of [39] over [44] is due to an additional stipulation such that, if a statement has been executed by at least one failed test case, then the total contribution from all the successful tests that execute the statement should be less than the total contribution from all the failed tests that execute it.

Table VI presents data on the total number of statements to be examined by using D^*, Crosstab, RBF, H3b, H3c, Tarantula, Ochiai, and Ochiai2, to locate faults in all the faulty versions of each subject program for both the best and worst cases. As seen from the data, these 7 techniques can be further divided into two groups based on their performance. Group I includes Tarantula, Ochiai, and Ochiai2, which are in general less effective than the other 4 techniques (Crosstab, RBF, H3b, and H3c) in Group II.

With respect to Group I, the smallest * to make D^* more effective than Tarantula, Ochiai, and Ochiai2 is 2 (namely, starting from D^2) for all the programs (best and worst cases) in Table II with only one exception for the worst case of the *make* program, where the smallest * is 4. Moreover, in some cases, the worst case of D^2 is even better than the best cases of other techniques in Group I. For example, for the *grep*, *space*, and *flex* programs, the worst case of D^2 is also better than the best case of Tarantula and Ochiai2.

With respect to Group II, 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, for the best and worst cases of *gzip* and *sed*, D^* is more effective than others when * is equal to or greater than 2. For *space* and *Ant*, the smallest * is 3 and 5, respectively, no matter whether it is the best or the worst case; for *Siemens* (best and worst), it is 7; for the best case of *grep*, it is 130; for the worst case, it is 7; for *make* (best and worst), it is 30; for *flex* (best and worst), it is 115; for the best case of *Unix*, it is 70; and for the worst case, it is 35. Similarly, in some cases, the worst case

$$\frac{N_{CF} \times N_{US}}{\sqrt{(N_{CF} + N_{CS}) \times (N_{US} + N_{UF}) \times (N_{CF} + N_{UF}) \times (N_{CP} + N_{US})}}$$

of D^* is even better than the best cases of other techniques in Group II. For example, for the *Ant* program, the worst case of D^2 is also better than the best case of H3b and H3c.

Similar to Fig. 4, an EXAM score-based comparison for the *Unix* suite and the *gzip* and *sed* programs is presented in Fig. 6. From parts (a) and (b) of that figure, we find that, on the *Unix* program, by examining less than 10% of the code, D^{70} can locate faults for 81.39% of the faulty versions in the best case, and 44.18% in the worst. In contrast, by examining the same amount of code, RBF (the second best technique) can only locate 72.67% of the faults in the best case, and 43.60% in the worst. The percentage for H3c (the third best) is 68.60% in the best case, and 41.86% in the worst. The curves in parts (c) and (d) of Fig. 6 are the three techniques with the top performance for the *gzip* program, and those in parts (e) and (f) of that figure are the best three techniques for the *sed* program.

For the same reason explained in Section 3.5, only three curves are displayed in each part of Fig. 6 to assure clarity and readability.[5] In summary, with respect to the EXAM score, D^* (with an appropriate value for the *) performs better than other techniques. This result is consistent with our observations from Table VI that D^* is the most effective technique.

The Wilcoxon signed-rank test was also conducted. The confidence to accept the alternative hypothesis (D^* is more effective than the other techniques) in many cases is more than 90%, regardless of the subject programs, or the best or worst case.

Considering all the data, it clearly indicates the superiority of D^* (with an appropriate value for the *) over the other techniques in locating program bugs.

VI. Programs With Multiple Faults

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

A. The Expense Score-Based Approach

For programs with multiple faults, the authors of [48] define an evaluation metric, *Expense*, corresponding to the percentage 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 the *same* as the EXAM score used in this paper (see Section 3.3.1).

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 * equal to 2 and 3 in this case) and other fault localization techniques using this strategy. As per [48], all the techniques are evaluated on the basis of the expense required to find only the first fault in the ranking. Note that because 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 to those with 5 faults. To evaluate the effectiveness, we compare $D^3$and D^2 against all the similarity coefficient-based techniques in Table III, and also all the contemporary fault localization techniques discussed in Section 5.

Data in Table VII gives the total number of statements that need to be examined to find the first fault across all 75 faulty versions. We observe that, regardless of best or worst case, D^3 is the most effective, and D^2 is the third (behind D^3 and RBF) among all the competing techniques.

Part (a) of Fig. 7 presents the best case comparison using the Expense or EXAM score, while Part (b) shows the worst. We see that D^3 generally performs better than other techniques.

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 VIII. The data suggests that even for multi-fault programs it can be confidently claimed that D^3 is more effective than the other techniques, irrespective of best or worst case. Together with the data in Table VII, and the curves in Fig. 7, there is strong evidence for the superiority of D^* (with an appropriate value for the *) over all the techniques compared in this paper.

B. The One-Fault-at-a-Time Approach

The *Expense* score, defined in [48] as the percentage of code that needs to be examined to find only the first bug in a multi-fault program, is really part of a bigger process that involves locating and fixing all faults (that result in at least one test case failure) that reside in the subject program. After the first fault has successfully been located and fixed, the next step is to re-run test cases to detect subsequent failures, whereupon the next fault is located and fixed. The process continues until failures are no longer observed, and we conclude (but are not guaranteed) that there are no more faults present in the program.

This process is referred to as the one-fault-at-a-time approach, and thus the *Expense* score only assesses the fault localization effectiveness with respect to the first iteration of the process. Because each iteration involves locating a particular fault, there is an *Expense* score associated with every iteration, and the number of iterations would equal the number of faults. Essentially, the total number of statements examined to find k faults (i.e., the cumulative *Expense* score) can be quantified as $\sum_{i=1}^{k} \alpha_i$, where α_i is the number of statements examined in the ith iteration. Thus, two fault localization techniques may be compared with one another by virtue of the total number of statements examined (i.e., the cumulative *Expense* score), if the one-fault-at-a-time approach is followed.

To further illustrate this result via example, we randomly select a 5-fault version (a program with five faults in it) from among the 75 multi-fault versions generated for the *Siemens* suite (based on the experiments in Section 6.1), and show how

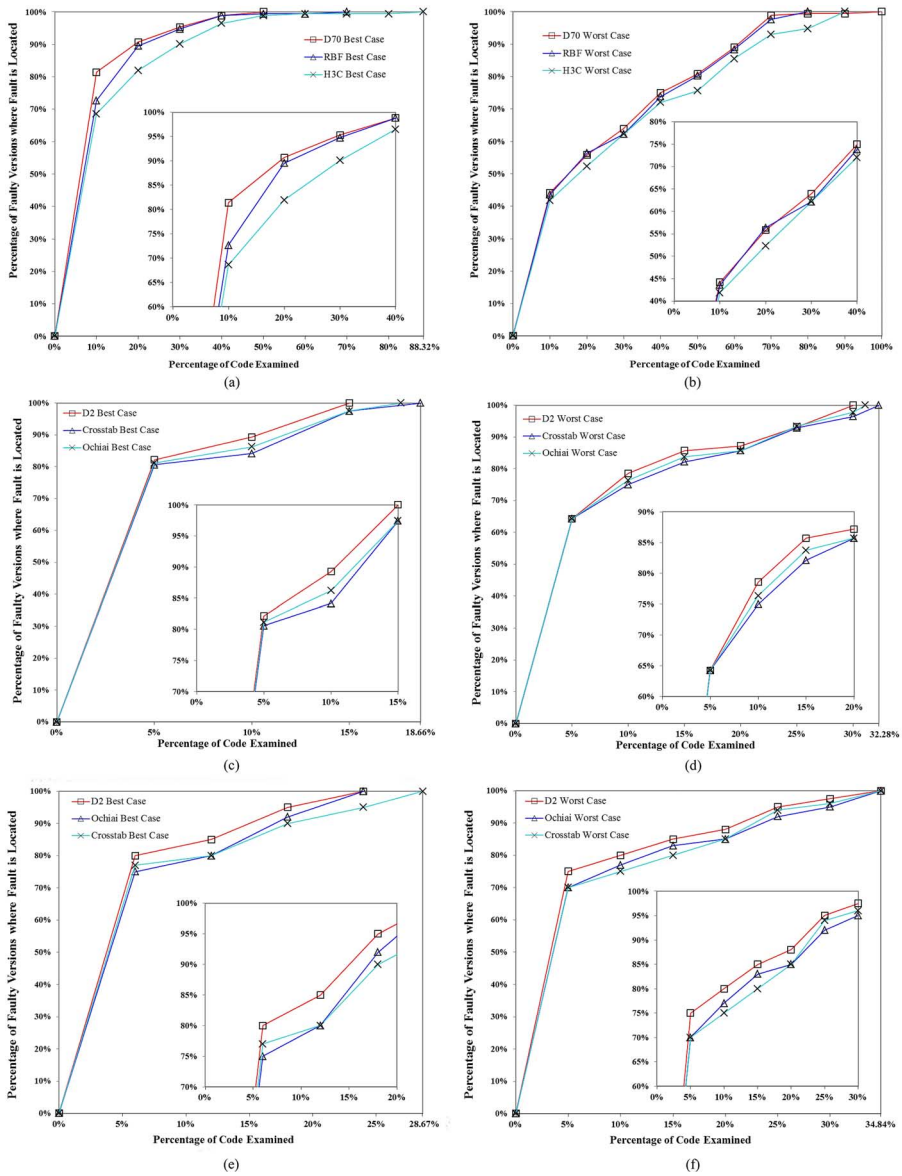

Fig. 6. EXAM score-based comparison between D^* and other contemporary fault localization techniques.(a) Best case on *Unix*, (b) Worst case on *Unix*, (c) Best case on *gzip*, (d) Worst case on *gzip*, (e) Best case on *sed*, (f) Worst case on *sed*.

D^3, D^2, and other techniques discussed in Sections 3 and 5 might be applied to locate all the faults sequentially.

Table IX reports the number of statements examined by each technique for both the best and worst effectiveness across each iteration, and its cumulative total. We observe that the total number of statements examined by D^3 is less than D^2, which is less than the other techniques for both the best and the worst cases. Also, except for a few exceptions, D^3 is more effective than or as effective as D^2 for each iteration, which is more effective than or as effective as the other fault localization techniques regardless of whether the best or worst case is considered.

VII. Discussion

In this section, we discuss interesting and relevant issues regarding D^*, as well as outline the potential threats to validity.

A. The Third Intuition Behind D*

Recall our third intuition behind the construction of D^*, as stated in Section 2.2, that asserts the suspiciousness assigned to a statement should be inversely proportional to the number of failed test cases that do not execute it.

Thus, we penalize a statement (view it as less suspicious) for every test case that fails without executing it. This intuition in general can be applied to programs with only one fault. However, what if there is more than one fault in the program? In such a situation, one might wonder if it makes sense to penalize a faulty statement by reducing its suspiciousness because of the existence of a test case that fails due to another faulty statement related to a different fault. There are three important factors to keep in mind. First, the objective of penalizing such statements is to bring down the suspiciousness of non-faulty statements. Second, the dominant factor in computing the statement suspiciousness using D^* is still N_{CF}, not N_{UF} (as per the suspiciousness function $(N_{CF})^*/(N_{UF}+N_{CS})$) because N_{CF} is the

TABLE VI
D* VERSUS CROSSTAB, RBF, H3B, H3C, TARANTULA, OCHIAI, AND OCHIAI2: TOTAL NUMBER OF STATEMENTS EXAMINED TO LOCATE ALL THE FAULTS

	Best Case									Worst Case								
	Unix	*Siemens*	*grep*	*gzip*	*make*	*Ant*	*space*	*flex*	*sed*	*Unix*	*Siemens*	*grep*	*gzip*	*make*	*Ant*	*space*	*flex*	*sed*
D^2	1805	1754	2596	1215	7689	672	1300	810	1787	5226	2650	4053	2681	13130	1184	2375	1493	3937
D^3	1667	1526	2528	1084	7676	368	1241	800	1714	5088	2422	3985	2535	13117	880	2316	1490	3864
D^4	1594	1460	2462	1083	7442	293	1200	790	1694	5015	2356	3919	2534	12883	805	2275	1489	3844
D^5	1507	1435	2391	1081	7442	228	1193	769	1675	4928	2331	3848	2532	12883	740	2270	1483	3825
D^6	1455	1400	2358	1081	7442	209	1190	768	1652	4876	2298	3815	2532	12883	721	2265	1482	3802
D^7	1423	1386	2336	1081	7394	196	1185	763	1635	4844	2284	3793	2532	12835	708	2260	1477	3785
D^8	1421	1362	2323	1081	7315	196	1178	762	1629	4840	2260	3780	2532	12756	708	2253	1476	3779
D^9	1418	1328	2285	1081	7315	181	1176	762	1626	4838	2226	3742	2532	12756	693	2251	1476	3776
D^{10}	1415	1327	2279	1081	7315	181	1147	762	1626	4835	2225	3736	2532	12756	693	2222	1476	3776
D*	1299 (*=70))		1704 (*=130)		6560 (*=30))			674 (*=115)		4757 (*=35)				12001 (*=30)			1393 (*=115)	
Crosstab	2524	2005	3486	1309	9409	1076	1790	1336	2068	6094	2873	6647	2760	14622	1578	2865	3184	4200
RBF	1302	2114	2015	2961	8908	233	1295	695	2977	4758	2980	3796	4412	13947	759	2375	1395	5074
H3b	1763	1439	2662	1530	8524	1358	2224	818	2905	5136	2335	4118	2982	13737	1860	3299	1532	5037
H3c	1717	1396	2345	1530	6911	1320	2199	692	2842	5090	2292	3801	2981	12124	1822	3274	1406	4974
Tarantula	3394	2453	5164	3105	13415	5964	3608	6527	4061	7704	3311	6906	4605	19467	9935	4826	7683	6655
Ochiai	1906	1796	2665	1265	7704	887	1369	840	1829	5322	2692	4121	2716	12917	1389	2444	1554	3961
Ochiai2	3345	2733	5075	1359	11701	1131	3392	2709	3271	7172	3726	8449	2810	16914	1633	4467	4776	5403

Note: The cell with a black background gives the smallest value of * such that D* outperforms all the competing techniques.

TABLE IX
NUMBER OF STATEMENTS THAT NEED TO BE EXAMINED FOLLOWING THE ONE-FAULT-AT-A-TIME APPROACH

		Number of statements examined in each iteration					Cumulative Total
		1	2	3	4	5	
D^3	Best	1	8	10	32	2	53
	Worst	3	11	10	41	2	67
D^2	Best	1	2	20	77	2	102
	Worst	3	5	20	86	2	116
Braun Banquet	Best	1	32	86	103	5	227
	Worst	3	35	86	112	5	241
Dennis	Best	1	2	79	104	5	191
	Worst	3	5	79	113	5	205
Mountford	Best	1	3	61	99	5	169
	Worst	3	6	61	108	5	183
Fossum	Best	1	6	52	82	2	143
	Worst	3	9	52	91	2	157
Pearson	Best	1	2	61	99	2	165
	Worst	3	5	61	108	2	179
Gower	Best	1	24	73	193	164	455
	Worst	4	26	76	193	172	471
Michael	Best	2	1	25	103	2	133
	Worst	4	4	25	112	2	147
Pierce	Best	1	33	105	108	89	336
	Worst	3	36	223	227	178	667
Baroni-Urbani&Buser	Best	1	3	96	116	27	243
	Worst	3	6	96	125	27	257
Tarwid	Best	2	33	87	104	5	231
	Worst	4	36	87	113	5	245
Ample	Best	1	2	65	115	2	185
	Worst	3	5	65	124	2	199
Phi (Geometric Mean)	Best	1	2	61	99	2	165
	Worst	3	5	61	108	2	179
Arithmetic Mean	Best	1	2	61	99	5	168
	Worst	3	5	61	108	5	182
Cohen	Best	1	2	85	104	5	197
	Worst	3	5	85	113	5	211
Fleiss	Best	1	2	104	124	5	236
	Worst	3	5	104	133	5	250
Zoltar	Best	6	112	1	112	2	233
	Worst	15	114	223	114	5	471
Harmonic Mean	Best	1	2	61	99	2	165
	Worst	3	5	61	108	2	179
Rogot2	Best	1	2	61	99	2	165
	Worst	3	5	61	108	2	179

		Number of statements examined in each iteration					Cumulative Total
		1	2	3	4	5	
Simple Matching	Best	1	59	104	124	63	351
	Worst	3	62	104	133	63	365
Rogers & Tanimoto	Best	1	59	104	124	63	351
	Worst	3	62	104	133	63	365
Hamming	Best	1	59	104	124	63	351
	Worst	3	62	104	133	63	365
Hamann	Best	1	59	104	124	63	351
	Worst	3	62	104	133	63	365
Sokal	Best	1	59	104	124	63	351
	Worst	3	62	104	133	63	365
Scott	Best	1	3	104	124	27	259
	Worst	3	6	104	133	27	273
Rogot1	Best	1	3	104	124	27	259
	Worst	3	6	104	133	27	273
Kulcynzki	Best	1	2	76	103	5	187
	Worst	3	5	76	112	5	201
Anderberg	Best	1	2	76	103	5	187
	Worst	3	5	76	112	5	201
Dice	Best	1	2	76	103	5	187
	Worst	3	5	76	112	5	201
Goodman	Best	1	2	76	103	5	187
	Worst	3	5	76	112	5	201
Jaccard	Best	1	2	76	103	5	187
	Worst	3	5	76	112	5	201
Sorensen-Dice	Best	1	2	76	103	5	187
	Worst	3	5	76	112	5	201
Crosstab:	Best	1	2	61	99	2	165
	Worst	3	5	61	108	2	179
RBF	Best	1	2	61	79	5	148
	Worst	3	5	61	88	5	162
H3b	Best	8	112	121	167	2	410
	Worst	17	112	123	167	5	424
H3c	Best	7	116	122	167	2	414
	Worst	16	116	124	167	5	428
Tarantula	Best	2	33	87	104	5	231
	Worst	4	36	87	113	5	245
Ochiai	Best	1	2	49	79	2	133
	Worst	3	5	49	88	2	147
Ochiai2	Best	1	2	83	116	5	207
	Worst	3	5	83	125	5	221

term with an exponent, not N_{UF}. Thus, the suspiciousness assigned to a statement has much more to do with N_{CF} than N_{UF}. Third and finally is the fact that the intuition holds empirically because of the superior performance of D* compared to the other fault localization techniques, regardless of whether the programs have just one fault or multiple faults, as per the evaluation in Section 6. This fact allows us to be confident in the validity and usefulness of this intuition, and the suspiciousness computation of D* as a whole.

B. Best and Worst Effectiveness Versus Other Alternatives

Per the discussion towards the beginning of Section 3.5, in the event of ties in the statement ranking, there may be two different levels of effectiveness: the best, and the worst. Consequently, the approach in this paper has been to provide data with respect to both levels of effectiveness. In contrast, some other fault localization studies follow a different approach. For example, in [23] the convention has been to report only the worst case effectiveness. This is a reasonable approach in that it is conservative. However, the problem lies with the fact that leaving out information on the best case effectiveness does not allow any estimate on where the actual effectiveness is expected to be (some-

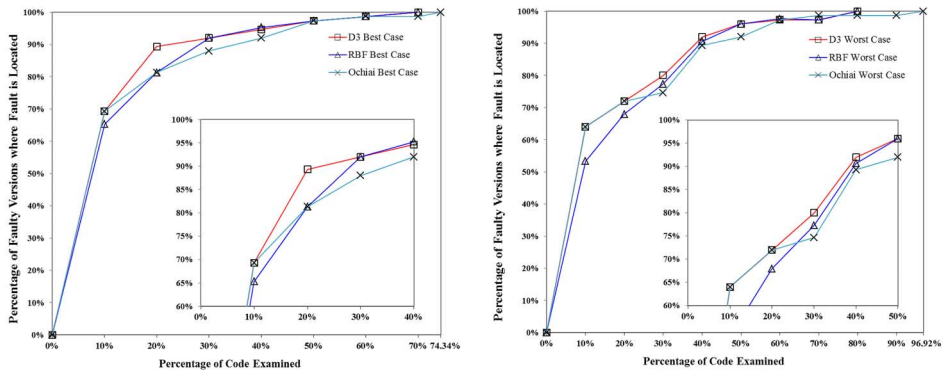

Fig. 7. Effectiveness comparison for the 75 multi-fault versions of the *Siemens* suite using the Expense/EXAM score. (a) Best case, (b) Worst case.

TABLE VII
TOTAL NUMBER OF STATEMENTS EXAMINED FOR THE 75 MULTI-FAULT VERSIONS OF THE *SIEMENS* SUITE BY D*, AND OTHER FAULT LOCALIZATION TECHNIQUES

Technique	Best Case	Worst Case
D^3	884	1327
D^2	932	1374
Braun Banquet	1449	1864
Dennis	1334	1736
Mountford	1135	1546
Fossum	1022	1427
Pearson	1648	2037
Gower	2619	2926
Michael	1439	1858
Pierce	2037	3099
Baroni-Urbani&Buser	1635	1991
Tarwid	1451	1853
Ample	1782	2153
Phi (Geometric Mean)	1191	1601
Arithmetic Mean	1140	1553
Cohen	1388	1782
Fleiss	1898	2283
Zoltar	2088	2555
Harmonic Mean	1225	1620
Rogot2	1225	1620
Simple Matching	1870	2231
Rogers & Tanimoto	1870	2231
Hamming	1870	2231
Hamann	1870	2231
Sokal	1870	2231
Scott	1786	2154
Rogot1	1786	2154
Kulcynzki	1314	1717
Anderberg	1314	1717
Dice	1314	1717
Goodman	1314	1717
Jaccard	1314	1717
Sorensen-Dice	1314	1717
Crosstab	1191	1601
RBF	888	1334
H3b	2888	3390
H3c	2183	2665
Tarantula	1451	1851
Ochiai	976	1385
Ochiai2	1933	2352

TABLE VIII
CONFIDENCE WITH WHICH IT CAN BE CLAIMED THAT D^3 IS MORE EFFECTIVE THAN OTHER TECHNIQUES (BEST AND WORST CASES) BASED ON THE EXPENSE OR EXAM SCORE

Technique	Best Case	Worst Case
D^2	98.67%	98.66%
Braun Banquet	99.99%	99.99%
Dennis	99.99%	99.97%
Mountford	99.70%	99.09%
Fossum	99.99%	99.87%
Pearson	99.99%	99.99%
Gower	99.99%	99.99%
Michael	99.99%	99.99%
Pierce	99.99%	99.91%
Baroni-Urbani&Buser	99.90%	98.86%
Tarwid	99.99%	99.98%
Ample	99.99%	99.98%
Phi (Geometric Mean)	99.90%	99.81%
Arithmetic Mean	96.00%	96.16%
Cohen	99.99%	99.89%
Fleiss	99.99%	99.97%
Zoltar	94.33%	98.31%
Harmonic Mean	99.89%	99.74%
Rogot2	99.89%	99.74%
Simple Matching	99.99%	99.93%
Rogers & Tanimoto	99.99%	99.93%
Hamming	99.99%	99.93%
Hamann	99.99%	99.93%
Sokal	99.99%	99.93%
Scott	99.98%	99.58%
Rogot1	99.98%	99.58%
Kulcynzki	99.99%	99.98%
Anderberg	99.99%	99.98%
Dice	99.99%	99.98%
Goodman	99.99%	99.98%
Jaccard	99.99%	99.98%
Sorensen-Dice	99.99%	99.98%
Crosstab	99.90%	99.27%
RBF	91.11%	96.62%
H3b	97.92%	99.82%
H3c	95.28%	98.79%
Tarantula	99.98%	99.92%
Ochiai	98.41%	96.31%
Ochiai2	99.99%	99.99%

where between the best and worst effectiveness; quite possibly the average of these two effectiveness levels).

That being said, one might be tempted to just present data on the average effectiveness (computed by simply taking the arithmetic mean of the best and worst effectiveness). However, this approach is also not flawless because it results in a complete loss of information with respect to the variability of the observed effectiveness. To better understand this approach, consider a scenario where a fault can be located using some technique by examining six statements in the best case and eight statements in the worst, meaning that on average the fault can be located by examining seven statements. In this case, the variability is small, as both in the best and worst cases we are only one statement away from the average. But what would happen if another technique allows the fault to be located by examining only one statement in the best case and thirteen in the worst? On average, the fault can still be located by examining seven statements. Thus, the two techniques are equal with respect to their average effectiveness, even though they are drastically different in terms of their variability.

To avoid such loss of information, we opted against presenting data on just the worst case or average effectiveness alone, and instead present data corresponding to both the best and the worst case effectiveness.

C. Threats to Validity

The three evaluation metrics or criteria used in this paper (see Section 3.3) represent a threat to construct validity. While such metrics are suitable measures of fault localization effectiveness, by themselves they do not provide a complete picture of the effort spent in locating faults as developers may not examine statements one at a time, and may not spend the same amount of time examining different statements. We also assume that, if a developer examines a faulty statement, they will identify the corresponding fault(s). By the same token, a developer will not identify a non-faulty statement as faulty. If such perfect bug detection does not hold in practice, then the examination effort may increase. However, such concerns apply to all fault localization techniques, not just D*.

Given that the evaluation of the effectiveness of D* has been empirical, arguably our results may not be generalized to all programs. However, we took steps to counter this threat by employing nine sets of subject programs (24 different programs in all) to make our evaluation more comprehensive. We also ensured that the programs varied greatly in terms of size (we made use of small, mid, and large-sized programs), functionality, number of test cases, etc. In addition to the faulty versions that were downloaded with the subject programs, we created additional versions with injected faults (Section 3.1) to augment and enlarge the data set. This approach allows us to have confidence with respect to the applicability of D* to different programs, as well as in its ability to provide high quality fault localization effectiveness. Admittedly, the fact that a small number of faulty versions for some of the subject programs were created by us may represent a threat to internal validity. However, the additional faults were created using mutation-based fault injection, which has been shown to be an effective approach to

simulating realistic faults that can be used in software testing research to yield trustworthy results [4], [11], [25], [28].

Finally, we emphasize that, as with all such studies, the coverage measurement tools, compilers, operating systems, hardware platforms, etc., may also have an impact on the overall results. However, we have discussed our experimental design in detail (Sections 3.1–3.4), which serves to promote repeatability of experiments and reproducibility of results.

VIII. Related Work

We now overview fault localization studies, in addition to the ones already discussed in the previous sections, directing readers interested in further details to the accompanying references or our report, *A Survey on Software Fault Localization* [38], which presents a comprehensive review of the current fault localization state-of-art.

Most of the earlier debugging effort relied on slicing-based techniques [2], [3], [26], [36] to help programmers reduce the search domain to quickly locate bugs.

In [31], 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. Otherwise, the technique continues by constructing a program dependence graph and checking adjacent un-checked nodes in the graph until the bug is located. Also presented in [31] 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, focusing on the code 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 executed by all successful tests, but not by the failed test case.

Cleve and Zeller [8] 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 [49]. A problem of this approach is that the cost can be very high because 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.

More recent research emphasizes suspiciousness ranking-based techniques to sort statements in descending order of their likelihood of containing bugs and examine them one-by-one from the top. All the techniques used in our case studies are in this category.

As per [23], Tarantula performs better than techniques such as nearest neighbor, set union, set intersection, and cause transitions on the *Siemens* suite. Because D* is significantly more effective than Tarantula, D* is also expected to outperform the above techniques. In [6], 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 [22], 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 [50].

In [34] the original Tarantula technique [23] is replaced by Ochiai (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.

IX. 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 24 different programs, D* is shown to be more effective than 38 other fault localization techniques (including 31 similarity coefficient-based techniques (Table III), and 7 other contemporaries (Crosstab [42], RBF [40], H3b and H3c [39], Tarantula [23], Ochiai [1], [29], and Ochiai2 [27], [29]). This is a significant extension compared with an earlier version of this study [41], which only used 16 different techniques on 21 programs. Furthermore, the superior effectiveness of D* is not limited to programs containing just one fault; based on our investigation, it extends equally well to programs with multiple faults.

The relationship between the effectiveness of D* and the value of * is empirically examined; it is discovered that the effectiveness increases along with * before leveling off after * passes a critical point. Details of this upper bound on effectiveness are also carefully addressed.

Currently, a major ongoing study is underway to apply D* to programs with multiple bugs. Instead of terminating the localization process after the first fault is located (Section 6.1), or using the one-fault-at-a-time approach (Section 6.2), a Fuzzy C-means-based fault localization technique is proposed to group failed tests into different fault focused clusters such that all the failed tests from the same cluster, along with all the successful tests, can be used to locate the corresponding causative bug. We also wish to compare the effectiveness and efficiency of D* against other techniques which require different types of information. Care must be taken to avoid providing more information to one technique and giving it an unfair advantage over others in terms of more effectively or efficiently locating program bugs.

Appendix

One potential problem of the approach in [47] is that it does not consider the statements in the $\mathcal{SAME}$ set. Consider the following scenario. The program to be debugged has a fault in statement s_8. The ranking generated by using technique α is $s_{16}, s_{12}, s_{11}, s_5, s_7, s_2, s_6, s_8, s_1, s_{10}, s_{13}, s_{15}, s_4, s_{14}, s_9, s_3$, where the first five statements are in $\mathcal{GREATER}^{(\alpha)}$, the next three (namely, s_2, s_6, and s_8) in $\mathcal{SAME}^{(\alpha)}$, and the remaining eight in $\mathcal{LESS}^{(\alpha)}$. A different technique β is also used to generate the same ranking order but with different suspiciousness values such that the first six statements are in $\mathcal{GREATER}^{(\beta)}$, the next three (namely, s_6, s_8, and s_1) are in $\mathcal{SAME}^{(\beta)}$, and the remaining seven are in $\mathcal{LESS}^{(\beta)}$. According to [47], technique α is definitely more effective than technique β because $\mathcal{GREATER}^{(\alpha)} \subseteq \mathcal{GREATER}^{(\beta)}$ and $\mathcal{LESS}^{(\beta)} \supseteq \mathcal{LESS}^{(\alpha)}$. However, this may not always be the case. For example, of the three statements with the same suspiciousness, a programmer may examine s_8 after s_2 and s_6 using the ranking generated

by α and inspect a total of $5 + 3 = 8$ statements, whereas a different programmer may examine s_8 before s_6 and s_1 using the ranking generated by β and inspect a total of $6 + 1 = 7$ statements. Thus, the total number of statements examined by technique β is less than that by technique α. This result is contradictive to the previous conclusion. To overcome this problem, authors of [47] impose an additional constraint such that s_6 and s_8 must be examined in the same order. Even with such an unrealistic assumption, the number of statements examined by technique α can only be equal to (but not less than) that examined by technique β. Hence, the contradiction still holds. Nevertheless, this problem will not occur if we consider statements in the $\mathcal{SAME}$ set as well by comparing the best and the worst cases separately. Using the same example, technique α examines $5 + 1 = 6$ statements for the best case and $5 + 3 = 8$ for the worst, whereas technique β examines $6 + 1 = 7$ statements for the best and $6 + 3 = 9$ for the worst. Hence, α is more effective than β for both the best and the worst cases, which is consistent with the aforementioned conclusion.

REFERENCES

[1] R. Abreu, P. Zoeteweij, R. Golsteijn, and A. J. C. van Gemund, "A practical evaluation of spectrum-based fault localization," *J. Syst. Software*, vol. 82, no. 11, pp. 1780–1792, 2009.

[2] H. Agrawal, R. A. DeMillo, and E. H. Spafford, "Debugging with dynamic slicing and backtracking," *Software—Practice Exp.*, vol. 23, no. 6, pp. 589–616, Jun. 1996.

[3] H. Agrawal, J. R. Horgan, S. London, and W. E. Wong, "Fault localization using execution slices and dataflow tests," in *Proc. 6th Int. Symp. Software Rel. Eng.*, Toulouse, France, Oct. 1995, pp. 143–15.

[4] J. H. Andrews, L. C. Briand, and Y. Labiche, "Is mutation an appropriate tool for testing experiments?," in *Proc. 27th Int. Conf. Software Eng.*, St. Louis, Missouri, USA, May 2005, pp. 402–411.

[5] A. Avizienis, J. C. Laprie, B. Randell, and C. Landwehr, "Basic concepts and taxonomy of dependable and secure computing," *IEEE Trans. Dependable Secure Comput.*, vol. 1, no. 1, pp. 11–33, 2004.

[6] B. Baudry, F. Fleurey, and Y. le Traon, "Improving test suites for efficient fault localization," in *Proc. Int. Conf. Software Eng.*, Shanghai, China, May 2006, pp. 82–91.

[7] S. Choi, S. Cha, and C. C. Tappert, "A survey of binary similarity and distance measures," *J. Systemics, Cybern. Inf.*, vol. 8, no. 1, pp. 43–48, Jan. 2010.

[8] H. Cleve and A. Zeller, "Locating causes of program failures," in *Proc. 27th Int. Conf. Software Eng.*, St. Louis, Missouri, USA, May 2005, pp. 342–351.

[9] Clover: A Code Coverage Analysis Tool for Java [Online]. Available: http://www.atlassian.com/software

[10] V. Debroy and W. E. Wong, "On the equivalence of certain fault localization techniques," in *Proc. 26th Annu. ACM Symp. Appl. Comput., (ACM SAC)*, TaiChung, Taiwan, Mar. 2011.

[11] H. Do and G. Rothermel, "On the use of mutation faults in empirical assessments of test case prioritization techniques," *IEEE Trans. Software Eng.*, vol. 32, no. 9, pp. 733–752, Sept. 2006.

[12] G. Dunn and B. S. Everitt, *An Introduction to Mathematical Taxonomy*. Cambridge, U.K.: Cambridge Univ. Press, 1982.

[13] B. S. Everitt, *The Analysis of Contingency Tables*. Boca Raton, FL, USA: Chapman & Hall, 1977.

[14] S. Fitzgerald, R. McCauley, B. Hanks, L. Murphy, B. Simon, and C. Zander, "Debugging from the student perspective," *IEEE Trans. Education*, vol. 53, no. 3, pp. 390–396, Aug. 2010.

[15] D. Freeman, *Applied Categorical Data Analysis*. New York, NY, USA: Marcel Dekker, 1987.

[16] A. L. Goel, "Software reliability models: Assumptions, limitations and applicability," *IEEE Trans. Software Eng.*, vol. 11, no. 12, pp. 1411–1423, Dec. 1985.

[17] L. A. Goodman, *The Analysis of Cross-Classification Data Having Ordered Categories*. Boston, MA, USA: Harvard Univ. Press, 1984.

[18] M. H. Hassoun, *Fundamentals of Artificial Neural Networks*. Cambridge, MA, USA: MIT Press, 1995.

[19] A. Meyer, A. Garcia, A. Souza, and C. Souza, "Comparison of similarity coefficients used for cluster analysis with dominant markers in maize (Zea mays L)," *Genet. Molecular Biol.*, vol. 27, no. 1, pp. 83–91, 2004.

[20] D. A. Jackson, K. M. Somers, and H. H. Harvey, "Similarity coefficients: Measures of co-occurrence and association or simply measures of occurrence?," *Amer. Naturalist*, vol. 133, no. 3, pp. 436–453, Mar. 1989.

[21] M. Jiang, M. A. Munawar, T. Reidemeister, and P. Ward, "Efficient fault detection and diagnosis in complex software systems with information-theoretic monitoring," *IEEE Trans. Dependable Secure Comput.*, vol. 8, no. 4, pp. 510–522, Jul./Aug. 2011.

[22] J. A. Jones, J. Bowring, and M. J. Harrold, "Debugging in parallel," in *Proc. 2007 Int. Symp. Software Testing Anal.*, London, U.K., Jul. 2007, pp. 16–26.

[23] J. A. Jones and M. J. Harrold, "Empirical evaluation of the Tarantula automatic fault-localization technique," in *Proc. 20th IEEE/ACM Conf. Automated Software Eng.*, Long Beach, CA, USA, Dec. 2005, pp. 273–282.

[24] B. Liblit, M. Naik, A. X. Zheng, A. Aiken, and M. I. Jordan, "Scalable statistical bug isolation," in *Proc. 2005 ACM SIGPLAN Conf. Program. Lang. Design Implement.*, Chicago, IL, USA, June 2005, pp. 15–26.

[25] C. Liu, L. Fei, X. Yan, J. Han, and S. P. Midkiff, "Statistical debugging: A hypothesis testing-based approach," *IEEE Trans. Software Eng.*, vol. 32, no. 10, pp. 831–848, Oct. 2006.

[26] J. R. Lyle and M. Weiser, "Automatic program bug location by program slicing," in *Proc. 2nd Int. Conf. Comput. Appl.*, Beijing, China, June 1987, pp. 877–883.

[27] L. Naish, H. J. Lee, and K. Ramamohanarao, "A model for spectra-based software diagnosis," *ACM Trans. Software Eng. Methodol.*, vol. 20, no. 3, pp. 11:1–11:32, Aug. 2011.

[28] A. S. Namin, J. H. Andrews, and Y. Labiche, "Using mutation analysis for assessing and comparing testing coverage criteria," *IEEE Trans. Software Eng.*, vol. 32, no. 8, pp. 608–624, Aug. 2006.

[29] A. Ochiai, "Zoogeographic studies on the soleoid fishes found in Japan and its neighboring regions," *Bull. Jap. Soc. Sci. Fish*, vol. 22, pp. 526–530, 1957.

[30] R. Lyman Ott, *An Introduction to Statistical Methods and Data Analysis*, 4th ed. Independence, KY, USA: Duxbury Press, Wadsworth Inc., 1993.

[31] M. Renieres and S. P. Reiss, "Fault localization with nearest neighbor queries," in *Proc. 18th Int. Conf. Automated Software En.*, Montreal, Canada, Oct. 2003, pp. 30–39.

[32] The Siemens Suite (retrieved August 2006) [Online]. Available: www-static.cc.gatech.edu/aristotle/Tools/subjects

[33] The Software Infrastructure Repository (retrieved October 2008) [Online]. Available: http://sir.unl.edu/portal/index.html

[34] R. Santelices, J. A. Jones, Y. Yu, and M. J. Harrold, "Lighweight fault-localization using multiple coverage types," in *Proc. 31st Int. Conf. Software Eng.*, Vancouver, BC, Canada, May 2009, pp. 56–66.

[35] I. Vessey, "Expertise in debugging computer programs," *Int. J. Man Mach. Studies: Process Anal.*, vol. 23, no. 5, pp. 459–494, 1985.

[36] M. Weiser, "Programmers use slices when debugging," *Commun. ACM*, vol. 25, no. 7, pp. 446–452, July 1982.

[37] P. Willett, "Similarity-based approaches to virtual screening," *Biochem. Soc. Trans.*, vol. 31, no. 3, pp. 603–606, Jun. 2003.

[38] W. E. Wong and V. Debroy, A Survey on Software Fault Localization Department of Computer Science, University of Texas at Dallas, Tech. Rep. UTDCS-45-09, Nov. 2009.

[39] W. E. Wong, V. Debroy, and B. Choi, "A family of code coverage-based heuristics for effective fault localization," *J. Syst. Software*, vol. 83, no. 2, pp. 188–208, Feb. 2010.

[40] W. E. Wong, V. Debroy, R. Golden, X. Xu, and B. Thuraisingham, "Effective software fault localization using an RBF neural network," *IEEE Trans. Rel.*, vol. 61, no. 1, pp. 149–169, Mar. 2012.

[41] W. E. Wong, V. Debroy, Y. Li, and R. Gao, "Software fault localization using DStar (D^*)," in *Proc. 6th IEEE Int. Conf. Software Security Rel. (SERE)*, Washington, D.C., June 2012, pp. 21–30.

[42] W. E. Wong, V. Debroy, and D. Xu, "Towards better fault localization: A crosstab-based statistical approach," *IEEE Trans. Syst., Man, Cybern.*, vol. 42, no. 3, pt. Part C, pp. 378–396, May 2012.

[43] W. E. Wong, J. R. Horgan, S. London, and A. P. Mathur, "Effect of test set minimization on fault detection effectiveness," *Software-Practice Exp.*, vol. 28, no. 4, pp. 347–369, Apr. 1998.

[44] W. E. Wong, Y. Qi, L. Zhao, and K. Y. Cai, "Effective fault localization using code coverage," in *Proc. 31st Annu. Int. Comput. Software Appl. Conf. (COMPSAC)*, Beijing, China, Jul. 2007, pp. 449–456.

[45] *χSuds User's Manual*. : Telcordia Technologies, 1998.
[46] M. Xie and B. Yang, "A study on the effect of imperfect debugging on software development cost," *IEEE Trans. Software Eng.*, vol. 29, no. 5, pp. 471–473, May 2003.
[47] X. Xie, T. Y. Chen, F. C. Kuo, and B. Xu, "A theoretical analysis of the risk evaluation formulas for spectrum-based fault localization," *ACM Trans. Software Eng. Methodol.*.
[48] Y. Yu, J. A. Jones, and M. J. Harrold, "An empirical study on the effects of test-suite reduction on fault localization," in *Proc. Int. Conf. Software Eng. (ICSE)*, Leipzig, Germany, May 2008, pp. 201–210.
[49] A. Zeller and R. Hildebrandt, "Simplifying and isolating failure-inducing input," *IEEE Trans. Software Eng.*, vol. 28, no. 2, pp. 183–200, Feb. 2002.
[50] Z. Zhang, W. K. Chan, T. H. Tse, B. Jiang, and X. Wang, "Capturing propagation of infected program states," in *Proc. 7th Joint Meeting Eur. Software Eng. Conf. ACM SIGSOFT Symp. Found. Software Eng. (FSE)*, Amsterdam, The Netherlands, Aug. 2009, pp. 43–52.

W. Eric Wong received his Ph.D. in computer science from Purdue University. He is currently a Professor, and the Director of International Outreach in Computer Science with the University of Texas, Dallas. He also has an appointment as a Guest Researcher from NIST (National Institute of Standards and Technology), an agency of the U.S. Department of Commerce. Prior to joining UTD, he was with Telcordia (formerly Bellcore) as a Project Manager for Dependable Telecom Software Development. Dr. Wong received the Quality Assurance Special Achievement Award from Johnson Space Center, NASA, in 1997. His research focus is on the technology to help practitioners develop high quality software at low cost. In particular, he is doing research in software testing, debugging, metrics, safety, and reliability. Dr. Wong is a Vice President of the IEEE Reliability Society, and the Founding Steering Committee Chair of the IEEE International Conference on Software Security and Reliability (SERE).

Vidroha Debroy received his B.S. degree in software engineering, M.S. degree in computer science, and Ph.D. degree in software engineering from the University of Texas, Dallas. He is currently a Software Testing Engineer at Microsoft. Dr. Debroy's research interests include software testing and fault localization, program debugging, and automated and semi-automated ways to repair software faults.

Ruizhi Gao received his B.S. degree in software engineering from Nanjing University. He is a Ph.D. candidate in the computer science department at the University of Texas, Dallas. His current research interests include software testing, fault localization, and program debugging.

Yihao Li received his B.S. degree in software engineering from East China Institute of Technology, and the M.S. degree in computer science from Southeastern Louisiana University. He is currently working toward a Ph.D. degree in computer science at the University of Texas, Dallas. His current research interests include software testing and fault localization, program debugging, and software risk analysis.