To appear in *Proceedings of the 11th International Conference on Quality Software (QSIC 2011)*,
IEEE Computer Society Press, Los Alamitos, CA (2011)

On Practical Adequate Test Suites for Integrated Test Case Prioritization and Fault Localization★*

Bo Jiang
The University of Hong Kong
Pokfulam, Hong Kong
bjiang@cs.hku.hk

W. K. Chan†
City University of Hong Kong
Tat Chee Avenue, Hong Kong
wkchan@cs.cityu.edu.hk

T. H. Tse
The University of Hong Kong
Pokfulam, Hong Kong
thtse@cs.hku.hk

***Abstract*—An effective integration between testing and debugging should address how well testing and fault localization can work together productively. In this paper, we report an empirical study on the effectiveness of using adequate test suites for fault localization. We also investigate the integration of test case prioritization and statistical fault localization with a postmortem analysis approach. Our results on 16 test case prioritization techniques and four statistical fault localization techniques show that, although much advancement has been made in the last decade, test adequacy criteria are still insufficient in supporting effective fault localization. We also find that the use of branch-adequate test suites is more likely than statement-adequate test suites in the effective support of statistical fault localization.**

***Keywords*—Debugging, testing, continuous integration**

I. INTRODUCTION

Program testing detects the presence of faults in programs. Simply knowing such presence is, however, inadequate: Developers also want to debug the program, that is, to locate the faults and fix them. Testing and debugging account for at least 30% of the total effort of a typical project [1]. They should be tightly integrated further to save costs.

How far does research advancement go by using some of the state-of-the-art test case prioritization techniques to identify high priority test cases and use them to conduct statistical fault localization? Previous studies have explored this integration problem [13]: If only a prefix of an ordered regression test suite is selected for execution, the code coverage statistics on the program under regression test achieved by different prioritized strategies (such as random and greedy [7]) may be different. Because statistical fault localization techniques [2][10][17][18][22][25][26] may use their corresponding execution statistics to pinpoint suspicious program entities, understanding the tradeoff due to the use test case prioritization strategies [7][8][12][16][20] on the effectiveness of fault localization techniques is crucial to the cost-effective integration of testing and debugging processes.

Jiang et al. [13] studied the effectiveness of using the prioritized test suites generated by different test case prioritization techniques in locating faults through statistical fault localization techniques. In terms of relative mean percentage of code examined to locate faults, they found that random ordering and the additional greedy strategy using statement as the code granularity level [7] can be less affected than the clustering-based and total strategies.

What is the probability of obtaining a test suite that is both *adequate* with respect to some testing criteria and *effective* with respect to some fault localization techniques? In other words, to what extent may we expect such an adequate test suite to be effective in assisting developers in locating faults? Can the list of suspicious statements that include the faulty (or the most fault relevant) statements fit into a panel on an IDE canvas easily providing that this suggestion is produced by a test suite that is deemed effective?

In this paper, we study these questions. We report the results of an empirical study that involves 16 test case prioritization techniques and four statistical fault localization techniques on 11 subject programs. We compare *ART* [12] and *Greedy* [7] test case prioritization strategies. *ART* represents a strategy that randomly selects test cases followed by resolving randomness through a coverage measure, whereas *Greedy* represents a strategy that selects test cases through a coverage measure followed by resolving tie cases randomly. These two strategies put opposite emphases on the same data available for test case prioritization. Their aggregated results help offset the ordering factor between random selection and coverage data in test suite prioritization.

Both branch (a.k.a. all-edges) adequacy and statement adequacy are *practical* coverage criteria, which can be applicable to widely-used complex programs [4], and typical code profiling tools such as `gcov` can measure such elements that have been covered. We have conducted an experiment

* This research is supported in part by the General Research Fund of the Research Grants Council of Hong Kong (project nos. 111410 and 717308) and a Strategic Research Grant of City University of Hong Kong (project no. 7008039).

† Contact author.

over 262 faulty programs, each of which runs 1000 branch-adequate test suites to reorder test cases; in turn, each reordered test case has been used to locate faults by four fault localization techniques individually. We have repeated the same procedure by systematically varying a portion of each reordered test suite.

Although many statistical fault localization research results and achievements have been obtained in the past decade, our results still show many surprises. First, branch-adequate test suites achieve higher probabilities than statement-adequate test suites in supporting effective fault localization. Nonetheless, these two *practical* test adequacy techniques are still insufficient to equip existing statistical fault localization techniques to be effective more often than not. Second, many existing test case prioritization techniques do not effectively integrate with the use of adequate test suites to assign the first priority to those test cases that are useful for statistical fault localization, even though the whole test suites are effective in locating faults within a particular threshold, say 1%, of the code examined. Third, but not the least, we observe that existing fault localization techniques are still ineffective in suggesting faulty statements within one debug screen in typical IDEs. These point to the same direction that the current state of integration between testing and debugging techniques are still unsatisfactory.

A preliminary version [11] of this paper has reported part of the results on the Siemens suite. This paper significantly enhances the preliminary version by studying statement-adequate test suites in additional to branch-adequate ones, extending the study with a comparison between *ART* and (Additional) *Greedy*, and reporting the results on a suite of four UNIX programs.

The main contribution of this paper and its preliminary version [11] is threefold: (1) It presents the first controlled experiment to study the probability of obtaining a test suite that is both *adequate* with respect to specific testing criteria and *effective* with respect to specific fault localization techniques. (2) It reports on how likely an average test case prioritization technique effectively supports an average statement-level statistical fault localization technique. (3) It conducts a regression analysis that shows that the mean effectiveness of statistical fault localization techniques is still not scalable as the size of a program increases.

We organize the rest of paper as follows: Section II reviews the test case prioritization techniques and fault localization techniques used in our study. We present our controlled experiment and its results in Section III. Section IV describes related work followed by a conclusion in Section V.

II. BACKGROUND

This section describes the test case prioritization and fault localization techniques involved in our study.

A. Test Case Prioritization Techniques

We follow [7] to organize the test case prioritization techniques into two dimensions. The first dimension is *granularity*, expressed in terms of *statements*, *branches*, and *functions*. The second is *prioritization strategy*. We study *Greedy* [7] and the *ART* [12] strategies. The *Greedy* strategy can be further subdivided into the *Total* and *Additional* sub-strategies. The *ART* strategy is reported in [12].

On one hand, *ART* represents a strategy that randomly selects test cases followed by resolving the randomness among the selected test cases through a coverage measure. On the other hand, *Greedy* represents a strategy that selects test cases through a coverage measure followed by resolving tie cases randomly. We refer to these two ways to prioritize test cases as the *random-before-coverage* (*R*2*C*) *strategy*, and the *coverage-before-random* (*C*2*R*) *strategy*, respectively. Table 1 summarizes the techniques.

*C*2*R strategy.* When we combine the two *Greedy* sub-strategies with the three granularities, we produce six techniques: total statement (total-st), total branch (total-br), total function (total-fn), additional statement (addtl-st), additional branch (addtl-br), and additional function (addtl-fn). All of them have been reported in [7].

TABLE 1. TEST CASE PRIORITIZATION TECHNIQUES.

	Ref	Type/Name		Brief Descriptions
	*T*1	Random		Random prioritization
Greedy (C2R)		**Greedy**		
	*T*2	total-st		Total statement
	*T*3	total-fn		Total function
	*T*4	total-br		Total branch
	*T*5	addtl-st		Additional statement
	*T*6	addtl-fn		Additional function
	*T*7	addtl-br		Additional branch
ART (R2C)		**ART**	**Level of Coverage Information**	**Test Set Distance**
	*T*8	ART-st-maxmin	Statement (*T*8-*T*10)	Maximize the minimum distance between test cases
	*T*9	ART-st-maxavg		Maximize the average distance between test cases
	*T*10	ART-st-maxmax		Maximize the maximum distance between test cases
	*T*11	ART-fn-maxmin	Function (*T*11-*T*13)	Maximize the minimum distance between test cases
	*T*12	ART-fn-maxavg		Maximize the average distance between test cases
	*T*13	ART-fn-maxmax		Maximize the maximum distance between test cases
	*T*14	ART-br-maxmin	Branch (*T*14-*T*16)	Maximize the minimum distance between test cases
	*T*15	ART-br-maxavg		Maximize the average distance between test cases
	*T*16	ART-br-maxmax		Maximize the maximum distance between test cases

*R*2*C Strategy.* We adopt the *ART* strategy [12] to represent the R2C strategy. The basic algorithm of *ART* prioritizes the test cases by iteratively building a candidate set of test cases, and then picks one test case out of the candidate set until all the test cases in a given regression test suite have been selected. To generate a candidate set of test cases, the algorithm randomly adds the not-yet-selected test cases one by

one into the candidate set (which is initially empty) as long as they can increase the code coverage achieved by the candidate set. The algorithm then selects a test case from the candidate set that maximizes the distance of the test cases from the selected test cases. The distance between two test cases is defined as the Jaccard distance between the coverage of the program entities of the two test cases. By combining three distance measures (average, minimum, and maximum) and the three granularities, there are nine techniques: ART-st-maxmin, ART-st-maxavg, ART-st-maxmax, ART-fn-maxmin, ART-fn-maxavg, ART-fn-maxmax, ART-br-maxmin, ART-br-maxavg, and ART-br-maxmax. All of them have been reported in [12].

Optimization. Readers may be aware that the above techniques have not been optimized. There are many ways to optimize the coverage-bases measures, such as through hill-climbing or genetic techniques. In the study reported in this paper, we do not examine the effects of optimization.

B. Fault-Localization Techniques

We revisit the set of four statistical fault localization techniques used in the study. Each technique computes the suspiciousness of individual statements, followed by ranking these statements according to their suspiciousness scores. One of the techniques, namely *Tarantula* [14], further uses a tie-breaker to resolve statements having the same suspiciousness values so that they may be assigned different ranks. This set of techniques were used in the experiment presented in [14].

TABLE 2. STATISTICAL FAULT LOCALIZATION TECHNIQUES.

Technique	Ranking formula
Tarantula [14]	$\frac{\%\text{failed}(s)}{\%\text{failed}(s) + \%\text{passed}(s)}$ Tie-breaker: $\max(\%\text{failed}(s), \%\text{passed}(s))$
Adapted Statistical Bug Isolation (SBI) [17]	$\frac{failed\ (s)}{passed\ (s)\ +\ failed\ (s)}$
Jaccard [2]	$\frac{failed\ (s)}{totalfailed\ +\ failed\ (s)}$
Ochiai [2]	$\frac{failed\ (s)}{\sqrt{totalfailed * (\ failed\ (s) + \text{passed}(s))}}$

Table 2 summarizes the fault localization techniques. In the table, the function *%failed* is the percentage of failed test cases that execute statement *s* (among all the failed test cases in the test suite). The function *%passed* is similarly defined. The function *failed* (*passed*, respectively) is the number of failed (passed, respectively) test cases for which *s* is executed. The variable *totalfailed* is the total number of failed test cases.

III. EXPERIMENT

In this section, we report the research questions and the setup of the experimental study.

A. Research Questions

We have designed two research questions to examine the more general questions stated in Section I.

RQ1: How likely does a *practical* test adequacy criterion generate adequate test suites for a statistical fault localization technique to locate faults effectively?

RQ2: To what extent may a test suite that is deemed effective in locating faults be prioritized so that the test cases having higher priority can be used for fault localization techniques to locate faults effectively?

We choose *branch coverage* (all-edges) and *statement coverage* to study RQ1 because they are practical criteria that can be applied to industrial-strength programs [4]. Moreover, many existing practical industrial-strength tools (such as gcov) can provide profiling data for testers to determine whether the coverage criteria have been achieved.

Answering RQ1 helps developers and researchers understand the chances of producing effective test suites based on practical test data adequacy criteria with respect to some of the best and representative statistical fault localization techniques. If the chance is high enough, developers can be more comfortable in using such adequate test suites to conduct regression testing on their programs so that the test data can be helpful for later and potential fault localization activities. However, if the chance is not good, developers are provided with evidence to support their actions to enhance their test suites for regression testing with a view to improving the chance of effective fault localization.

Answering RQ2 helps us decide whether the effort on prioritizing test cases is worthwhile and whether executing the higher priority portion of the prioritized test cases may still retain good fault localization effectiveness. If the finding is positive, developers may be comfortable in using the test data for fault localization. On the other hand, if the finding is negative, then additional test cases may be required so that the fault localization effectiveness of the test suites will not be seriously compromised.

B. Subject Programs and Test Suites

We use the *Siemens* suites and four UNIX programs as the subject (see Table 3). We have downloaded them from SIR [6]. The Siemens suite consists of seven small programs. Each program comes with, among other files, a set of faulty versions, a test pool, and a set of 1000 large branch-adequate test suites and 1000 large statement-adequate test suites. According to [6], each test case in every such brand-adequate (statement-adequate, respectively) test suite is randomly picked among test cases in the test pool that can cover the same edge (statement, respectively).

However, only one test pool is available to each UNIX program. We use this test pool to construct 1000 branch-adequate test suites and 1000 statement-adequate test suites for each UNIX program. More specifically, for each edge, we randomly pick a test case that covers the edge. We note that this is also the generation strategy used in the downloaded *large* test suites for the Siemens suite [6].

TABLE 3. SUBJECT PROGRAMS.

Group	Subject	No. of Faulty Versions	SLOC	Test Pool Size	No. of Test Suites
Siemens Suite	tcas	41	133–137	1608	1000
	schedule	9	291–294	2650	1000
	schedule2	10	261–263	2710	1000
	tot_info	23	272–274	1052	1000
	print_tokens	7	341–342	4130	1000
	print_tokens2	10	350–354	4115	1000
	replace	32	508–515	5542	1000
UNIX Programs	flex (2.4.7–2.5.4)	21	8571–10124	567	1000
					1000
	grep (2.2–2.4.2)	17	8053–9089	809	1000
					1000
	gzip (1.1.2–1.3)	55	4081–5159	217	1000
					1000
	sed (1.18–3.02)	17	4756–9289	370	1000
					1000

C. Metrics

To measure the fault localization effectiveness, we use the metric *Expense* [10], which is defined by the equation

$$Expense = \frac{rank\ of\ the\ faulty\ statement}{total\ number\ of\ executable\ statements},$$

where the rank of a given statement is the sum of the number statements that have higher suspiciousness values and the number of statements that have the same or higher tiebreaker values if their suspiciousness values are equal to that of the given statement.

In practice, a developer may only have the patience to walk through a small portion of the ranking list. As a result, a high Expense value (such as 90%) may be useless for debugging. A sequence of test cases with respect to a fault localization technique and a given faulty program is said to be ***α-effective*** if the Expense value of using this sequence of test cases by the fault localization technique on the faulty program is strictly lower than the threshold value specified by α.

We define the metric *Fault Localization Successful Percentage* (*FLSP*) to be the ratio of the number of ***α-effective*** test suites in a test suite pool P over the size of the test suite pool $|P|$ with respect to a fault localization technique and a given faulty program, thus:

$$FLSP(T, P, \alpha) = \frac{|\ t \mid t \in P \text{ and } t \text{ is } \alpha\text{-effective}\ |}{|P|}$$

D. Experimental Setup

We apply each test case prioritization technique (see Table 1) and each fault-localization technique (see Table 2) to every test suite of every subject program. For every prioritized test suite generated by each test case prioritization technique, we repeated the above procedure using, in turn, the top 10%, 20%, …, 90% of the ordered test suite. For each such portion of all prioritized test suites applicable to every corresponding subject, we collected the Expense values from all fault localization techniques, and computed the FLSP values.

We have carried out the experiment on a Dell PowerEdge 2950 server serving a Solaris UNIX system. We used gcc version 4.4.1 as the C compiler. The server has two Xeon 5430 (2.66GHz, 4 core) processors with 4GB physical memory. We follow [25] to remove those faulty versions that cannot detect by any test case in the test pool as well as those that can be detected by more than 20% of the test cases in the pool. We used gcov to collect the execution statistics of every run.

To study RQ1, we use all the branch-adequate and statement-adequate test suites for experimentation. For each faulty version, we also removed those test suites that cannot detect the fault because fault localization techniques require at least one failed test case. We have also removed all the test suites that cannot work with our platform. We pass the execution statistics to all the four fault localization techniques and follow [11] to measure their results in terms of FLSP on all subject programs with three different fault localization effectiveness threshold values (1%, 5%, and 10%). RQ2 is a follow-up research question based on the results of RQ1. We only use branch-adequate test suites for RQ2 to control the scale of our empirical study. Similar to RQ1, we have removed all test suites that contain no failed test cases as well as all test suites that cannot work with our platform.

All the *ART* techniques are based on random selection. Therefore, we follow [11] to repeat each of them 20 times to obtain an average performance and to select 50 suites from the available 1000 test suites for every Siemens or UNIX subject program. Thus, we conducted a total 1000 prioritizations for every *ART* technique. We then use MATLAB to perform multiple comparisons by specifying a significance level of 5% for analysis.

E. Data Analysis

1) Answering RQ1

We examine the effect of a fault localization technique to locate faults in programs using a whole adequate test suite. As a result, we need not differentiate among test case prioritization techniques, as the test suites generated by them will have the same fault localization results.

Table 4, Table 5, Table 6, and Table 7 show the mean number of effective suites averaged over all faulty versions for Siemens and UNIX programs on Tarantula, SBI, Jaccard, and Ochiai, respectively. The first row lists the threshold values used in the experiment. We use three threshold values to measure the effectiveness of fault localization results: 1%, 5%, and 10%. In other words, if a fault can be located by inspecting less than 1%, 5%, or 10% of the ranked list of suspicious statements, we consider the fault localization result to be useful for the respective scenarios. The threshold values divide the table into three groups. The second row shows two adequacy criteria (Br for branch adequacy and Stmt for statement adequacy) for each group. The rows that follow show the mean number of effective test suites for each program. (Note that the total number of test suites for each faulty version is 1000.)

TABLE 4. MEAN NUMBER OF EFFECTIVE TEST SUITES FOR TARANTULA

Threshold Value	α = 1%		α = 5%		α = 10%	
Adequacy Criteria	Br	Stmt	Br	Stmt	Br	Stmt
tcas	25	8	191	36	193	31
replace	145	46	342	126	381	140
tot_info	145	74	331	140	430	181
schedule	14	1	169	52	243	61
schedule2	0	0	35	5	107	11
print_tokens	55	3	151	22	215	37
print_tokens2	156	56	317	121	332	159
grep	618	371	832	546	936	756
sed	604	319	860	594	931	753
flex	665	376	827	586	951	720
gzip	674	395	843	516	945	711

We study how branch-adequate test suites compare with statement-adequate test suites. We observe from Table 4 that for every subject program and for each threshold value, on average, the use of a branch-adequate test suite performs consistently better than a statement-adequate test suite. Moreover, this result is consistent with the other three fault localization techniques as shown in Table 5, Table 6, and Table 7. Branch adequacy subsumes statement adequacy in terms of test requirement, and the former empirically outperforms the latter in terms of fault detecting ability. Our results show that branch-adequate test suites can also be more effective than statement-adequate test suites in supporting fault localization.

TABLE 5. MEAN NUMBER OF EFFECTIVE TEST SUITES FOR SBI

Threshold Value	α = 1%		α = 5%		α = 10%	
Adequacy Criteria	Br	Stmt	Br	Stmt	Br	Stmt
tcas	29	11	205	46	201	39
replace	152	48	353	134	388	154
tot_info	157	80	335	141	437	186
schedule	19	10	180	63	246	70
schedule2	0	0	38	15	109	16
print_tokens	61	5	153	35	215	43
print_tokens2	166	70	317	132	337	168
grep	690	306	831	595	952	742
sed	681	389	880	590	946	719
flex	678	306	860	541	940	754
gzip	611	324	818	548	953	759

TABLE 6. MEAN NUMBER OF EFFECTIVE TEST SUITES FOR JACCARD

Threshold Value	α = 1%		α = 5%		α = 10%	
Adequacy Criteria	Br	Stmt	Br	Stmt	Br	Stmt
tcas	31	9	235	41	217	37
replace	42	24	206	36	207	33
tot_info	153	56	354	138	394	154
schedule	147	80	338	158	433	186
schedule2	31	13	180	69	256	69
print_tokens	0	0	36	11	109	11
print_tokens2	73	16	156	32	216	51
grep	693	306	844	512	920	687
sed	660	389	799	513	885	747
flex	685	306	805	510	903	705
gzip	670	324	852	568	915	695

TABLE 7. MEAN NUMBER OF EFFECTIVE TEST SUITES FOR OCHIAI

Threshold Value	α = 1%		α = 5%		α = 10%	
Adequacy Criteria	Br	Stmt	Br	Stmt	Br	Stmt
tcas	31	15	198	49	201	26
replace	163	46	349	130	388	148
tot_info	157	76	338	138	448	181
schedule	12	8	180	61	243	64
schedule2	0	0	48	20	115	9
print_tokens	56	9	160	29	231	39
print_tokens2	168	58	320	123	343	173
grep	615	306	842	595	900	722
sed	669	389	805	574	925	716
flex	666	306	824	517	894	712
gzip	698	324	834	503	944	709

We have further conducted hypothesis testing to confirm that the use of a branch-adequate test suite is significantly more effective than a statement-adequate test suite. In a recent study [4], test suites with high branch coverage (95% on average) have been shown to be generable in a fully automated manner for complex programs. As such, we believe that the use of branch-adequate test suites is more promising than statement-adequate test suites for statistical fault localization.

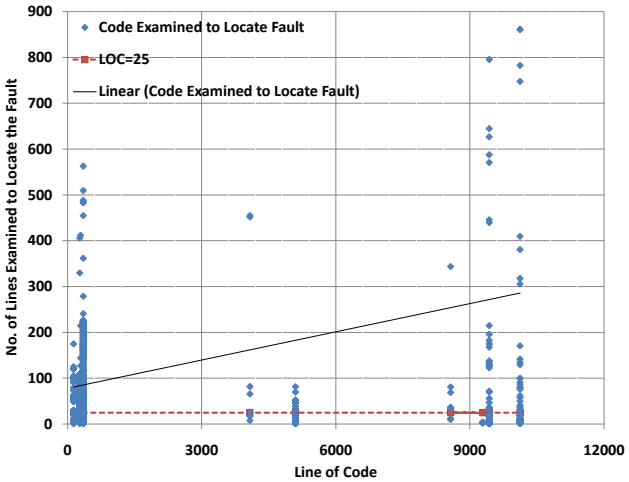


Fig. 1. Distribution of Expense vs. size of faulty programs.

The slope of the regression line is less than 0.013, indicating that branch- and statement-adequate test suites are increasingly practical in supporting effective statistical fault localization as the size of a faulty program increases. Nonetheless, most data points are above the dotted line, indicating that a typical 25-line screen in an IDE may be ineffective in displaying the code that includes the faults.

To understand how to support fault localization better, Fig. 1 plots all the data points in the experiment, each of which represents the Expense used to locate the fault in a program by a fault localization technique with different program sizes (measured in lines of code). The *x*-axis shows the lines of code for all the Siemens and UNIX program versions while the *y*-axis shows the number lines in the source code that need to be examined to find a fault. Each dot represents the code needed to locate a fault for a program with specific executable lines of code, and the solid line is a linear regression line for these dots, to show the trends of code examined to locate a fault.

Moreover, when using a fault localization tool, developers may expect it to help them focus their attention on only a few

suspicious source code locations; otherwise, in practice, the developers may lose patience and consider the tool ineffective. A typical debug screen of an IDE (such as an Eclipse IDE or Visual Studio IDE) is around 25 lines of code. By using this as a reference, one can draw a horizontal dotted line (in red) as shown in Fig. 1.

Interestingly, we find from Fig. 1 that the majority of the dots are distributed *above* the (red) horizontal dotted line, and the linear regression line is also positioned above the (red) dotted line. This observation shows that, in general, fault localization results based on existing adequate test suites cannot help developers locate the fault within one screen of the code view of a practical IDE.

Our analysis above is preliminary, and the aim of this analysis is not to completely answer whether existing IDEs can effectively present information on statistical fault localization results to developers. However, the finding does raise interesting questions for future work: Can we define test adequacy criteria that will likely construct test suites for effective fault localization that fit for one screen? Moreover, what information can fit into the code view on one screen to support effective fault localization (for practical adequate test suites)? Alternatively, what kinds of advances in human-computer interaction (such as screen design) will support effective fault localization of large applications?

Moreover, we find from Fig. 1 that the slope of the regression line is close to 0.013, and the line meets the y-axis at $y = 95$. In other words, the equation for the regression line is $y = 0.013x + 95$. It indicates that there are certain overheads in locating faults from programs of small sizes. The slope of the line (0.013) is small but larger than 0.01, which indicates a minimum of 1% of the code must be examined on average [24]. We observe that, in the literature, the use of 1% as the threshold is frequently reported in experiments that evaluate the effectiveness of statistical fault localization techniques. Interestingly, this benchmark requirement cannot be met on average even for the idealized scenarios that we have studied in this paper.

Our results indicate that existing practical test adequacy criteria are still unlikely to generate test suites to support effective statistical fault localization.

2) Answering RQ2

We have conducted a postmortem analysis on the integration results. Owing to the large number of possible criteria to specify whether an integration is effective, we use three different threshold Expense values, namely $\alpha = 0.01$, 0.05, and 0.10, as the criteria to deem a test suite to be effective. They represent the cases that developers need to examine up to 1%, 5%, and 10% of the code in order to find the faults if they follow the ranks of the statements. We will leave the analysis of different factors such as strategies and coverage granularity levels to be reported in future work.

a) Small-Scale Programs

Subfigures (a), (c), and (e) of Fig. 2 show the corresponding results. In each of the subfigures, the x-axis indicates different percentages of a test suite used for fault localization while the y-axis indicates the FLSP values for a test case prioritization technique to locate faults by examining up to the threshold percentage of code.

We observe from Fig. 2 (a) that, by inspecting the top 1% of the ranked list of statements, the median FLSP value of a test suite is 8% if we prioritize and execute the top 10% of a test suite for fault localization, which is very low. Even if we increase the percentage of test suite to 100%, the median of the percentages of effective test suites is still less than 14%. The result indicates that it is quite impractical to assume that the faults will be in the few (say, 1 to 5) top-ranked lines of source code.

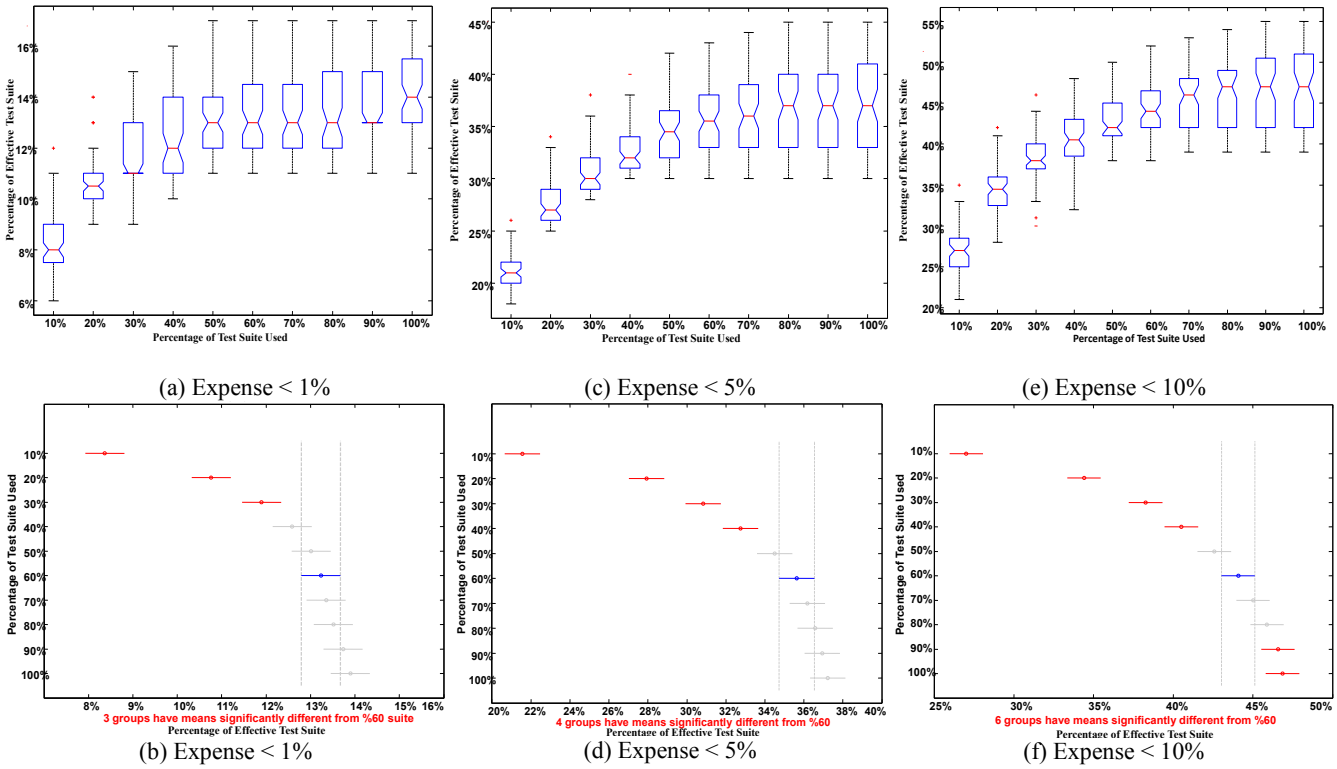
From Fig. 2 (a), (c), and (e), we observe that if a higher percentage of an original test suite is used for fault localization, the percentage of effective test suites increases. However, the increase is gradually less noticeable when the percentage of the test suite used reaches 60%. In particular given a code inspection range of 1%, the use of 60% of the prioritized test cases for the fault localization already achieves a FLSP value of 13%, whereas the use of all the remaining 40% of test cases will increase the *percentage* value to 14% only. We observe similar trends for code inspection ranges of 5% to 10% in Fig. 2 (c), and (e), respectively.

We have conducted ANOVA analysis to compare their mean FLSPs. The analysis results consistently reject the null hypothesis that the use of different percentages (namely, 10%, 20%, … , 100%) of the same ordered test suites has the same FLSP values at a significance level of 5%. To see what percentages of test suites differ from one another in terms of FLSP, we have further conducted the multiple comparisons procedure to find how different percentages of test suites differ significantly from one another at a significance level of 5%. Subfigures (b), (d), and (f) of Fig. 2 show the results. The solid lines not intersected by the two vertical lines represent the percentages of test suites whose means differ significantly from the use of 60% of the suite for fault localization, while the gray lines represent the percentages of test suites comparable to the use of 60% of the suites for fault localization.

From subfigures (b) and (d) of Fig. 2, we observe that executing 60% of a test suite has no significant difference from executing the entire test suite. If we relax the code examination range to 10% of the code for the Siemens suite, as shown in Fig. 2 (f), there will be a significant difference. It indicates that developers should have an estimate on the amount of code they can afford to examine so that a test case prioritization technique can use it as a reference to determine the portion of test suites to be executed.

a) Medium-Scale Programs

We have also conducted a postmortem analysis on the integration study for UNIX programs. Subfigures (a), (c), and (e) of Fig. 3 show the corresponding results. In these subfigures, the x-axis indicates different percentages of a test suite used for fault localization while the y-axis indicates the FLSP values for a test case prioritization technique to locate faults by examining up to the threshold percentage of code.



(a) Expense < 1% (c) Expense < 5% (e) Expense < 10%

(b) Expense < 1% (d) Expense < 5% (f) Expense < 10%

Fig. 2. The chance of test case prioritization techniques supporting effective fault localization for Siemens programs.
The chance of having all the branch-adequate test suites to support effective fault localization is not good. The use of a smaller portion of prioritized test suites does not support effective fault localization either.

We observe from Fig. 3 (a) that, by inspecting the top 1% of the ranked list of statements, the median FLSP value is 47% if we prioritize and execute the top 10% of a test suite for fault localization, which is much higher than that for the Siemens programs. Even if we increase the percentage of test suite to 100%, the median FLSP value is still under 65%. Although developers are willing to examine up to 5% (10%, respectively) of the code, Fig. 3 (c) and (e) still show that there is less than 65% (73%, respectively) of the chance that the top 10% of test cases can assist them in locating faults effectively. The results show that developers should not greedily start fault localization based on a small percentage (10% in the above discussion) of the whole test suite.

The data show that there are two strategies to alleviate this problem. First, we observe across Fig. 3 (a), (c), and (e) that, since the corresponding bars among the three plots increase in terms of their *y*-values, if developers are willing to put in more effort to examine the code, the effort may be worthwhile. Second, on each plot in Fig. 3 (a), (c), and (e), when a higher percentage of an original test suite is used for fault localization, the percentage of effective test suite increases remarkably. The results suggest that, if the preferred code examination range is fixed, the use of a higher percentage of test cases can be a good choice. It seems to us that this second strategy provides hints to answer the follow-up question in RQ1 that, in order to fit the code in supporting effective fault localization on one code-view screen, the use of a smaller adequate test suite for such testing-debugging integration may be a viable research direction. (However, the study on this aspect is not within the scope of this paper).

We have also conducted ANOVA analysis to compare the mean FLSPs. The analysis results consistently reject the null hypothesis that the use of different percentages of test suites has the same FLSP values at a significance level of 5%. We further conducted the multiple comparisons procedure to find how different percentages of the same ordered test suites differ significantly from one another at a significance level of 5%. Subfigures (b), (d), and (f) of Fig. 3 show the results. The solid lines not intersected by the two vertical lines represent those percentages of test suites whose means differ significantly from the use of 100% of the suite for fault localization, while the gray lines represents those percentages of test suites comparable to the use of 100% of the suites for fault localization.

From subfigure (b) of Fig. 3, we observe that only when executing more than 60% of a test suite will there be no significant difference from executing the entire test suite in terms of FLSP. If we relax the code examination range to 5% and 10% of the code as shown in subfigures (d) and (f) of Fig. 3, we still have the same results. It shows that, for UNIX programs, around 60% percentage of the test suite should be used to obtain fault localization effectiveness comparable to the use of the whole test suite. The results indicate that, by using smaller test suites, developers should prepare themselves that the fault localization effectiveness are extremely likely be decreased.

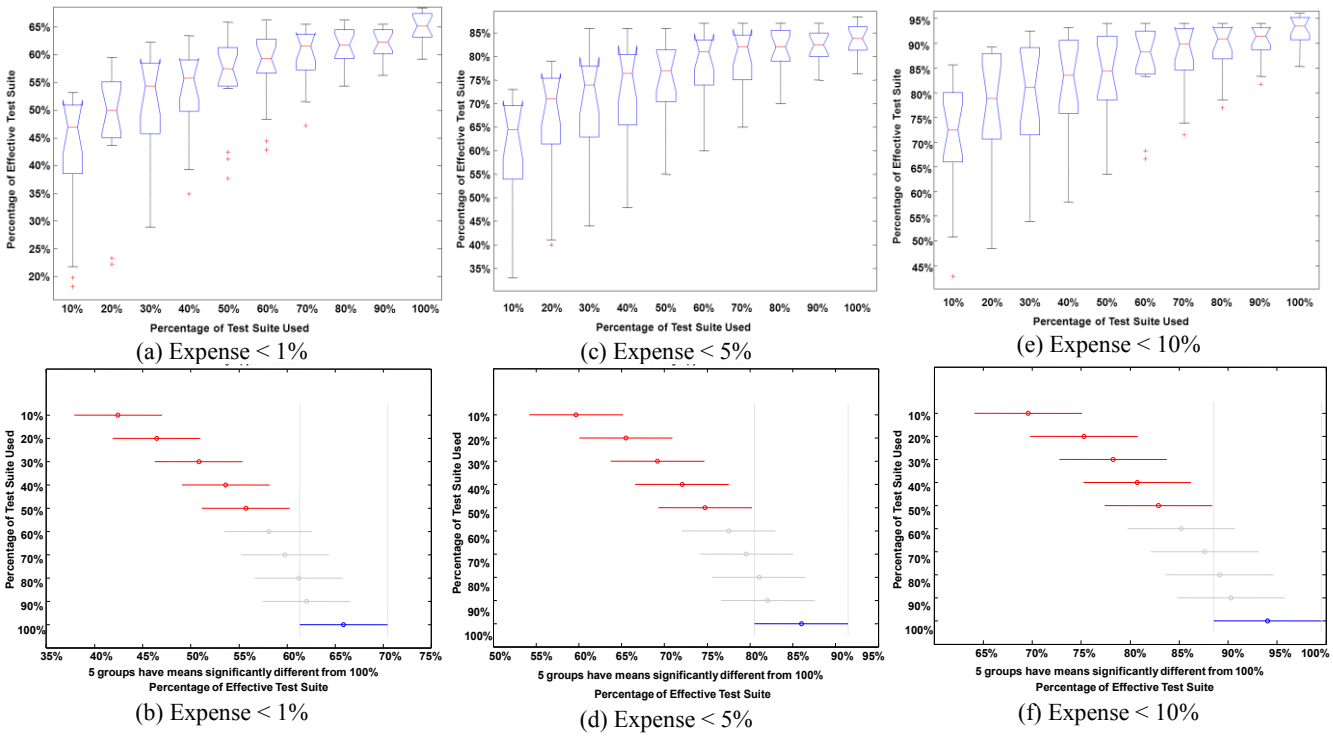


Fig. 3. The chance of test case prioritization techniques supporting effective fault localization for UNIX programs

To conclude, we can answer RQ2 that the chance of test case prioritization techniques to support effective fault localization is higher on medium-scale programs than on small-scale programs. Furthermore, similar to small scale programs, around 60% of the test suite should be used for medium-scale programs to make the selected part of the test suite as effective as the whole suite.

F. Threats to Validity

We have used seven Siemens programs, four UNIX programs, and their faulty versions as our subjects. The use of other faulty programs can result in different coverage patterns for failed test executions and passed test executions, which may result in different suspiciousness values assigned to the program statements. Although the set of faults did not represent all possible faults, using them to conduct comparisons among techniques published in existing work is useful for researchers to compare results across different papers and experiments. Moreover, we have used the adequate test suites provided by the SIR repository and generated the branch-adequate and statement-adequate test suites for the UNIX programs. The use of other adequacy test suites may provide other results. We will leave the analysis and reporting of such test suites in future work.

In any case, our subjects have been widely used in existing test case prioritization, statistical fault localization, and unit testing research. Moreover, branch-adequate test suites have been frequently used in the experiments of testing and debugging papers. We believe that they have used these subjects in their experiments on good grounds with practical considerations. The results of our experiment complemented their findings on these artifacts and facilitated comparison across publications.

In our experiment, we have excluded some faulty versions and test cases available from SIR. There are a few reasons. The foremost reason is that, in our testing framework for the experiment, it uses gcov, which is a popular and freely available tool, to collect the branch execution profile of each non-crashed execution. For crashed executions, gcov cannot provide coverage data. The techniques in our experiment, however, require coverage data in order to operate. Consequently, we have excluded these test cases from the data analysis. As we have reported, our experimental environment is a UNIX server running SUN OS. The C compiler provided by the underlying platform is also provided by SUN. Some versions cannot be compiled. This is a kind of platform dependence issue and we have also removed these versions to minimize their impact.

Another reason for us to exclude some faulty version from the data analysis is that we follow previous papers on test case prioritization to conduct the experiment to exclude any version whose failures can be detected by more than 20% of the test cases in the test pool. The choice of this threshold poses a threat to this study. Nonetheless, this practice has been widely used in the test case prioritization experiments. The use of this threshold facilitates a comparison between this work and existing publications. A way to address this threat could be to conduct a larger experiment to vary this threshold from 0% to 100% systematically, and observe the effect. The effort to conduct this experiment and the corresponding data analysis are, however, not affordable to us. We have, therefore, excluded this aspect from our current experiment.

Another concern about the study may be the nature of the test suites. We have used the test suites provided by SIR. They may not be representative in the sense that some test cases important to statistical fault localization may not be available. On the other hand, test case prioritization and fault localization are becoming mature and hence a common ground for comparison is necessary. To strike a balance between the use of more test suites and the comparability with a large body of published work, we have chosen the latter option in this study. In RQ1, we have 1000 branch-adequate and 1000 statement-adequate test suites for each subject program. They provide us enough data points to compile statistical results shown in the paper. For RQ2, we would like to highlight that the results are based on one small test pool per subject program. Readers should not overly generalize the results. For some subject programs, the requirement of having branch-adequate test suites may still be too demanding. For instance, almost all the subject programs used in the experiment reported in [4] did not come with test suites that are branch adequate. We leave this practical consideration as future work.

In this study, owing to time and resource constraints, we have only evaluated the random, the coverage-based *Greedy*, and the white-box *ART*-based test case prioritization techniques. Although they are among the best general test case prioritization techniques studied in previous work, they have not been optimized. The use of optimized versions or other variants of these strategies as well as the use of other strategies may produce different results.

In drawing a comparison, we use the Expense metric as well as the FLSP metric. The use of other metric may produce different results. The former metric has been widely used to evaluate statistical fault localization techniques. It, however, only represents one way of how developers may use the ranked list of statements and makes an assumption that any fault on each visited statement can be identified correctly. The time taken to visit such a statement and the precision of the fault identification has not been captured by this metric. The FLSP metric is built on top of the Expense metric. Owing to the limitation of the Expense metric, the effort to reveal a fault measured by the FLSP metric does not totally reflect the effort of developers to use the generated ranked list of statements to perform debugging. Readers are advised to interpolate the results of the experiment carefully.

IV. Related Work

Apart from the 16 test case prioritization techniques and the 4 fault localization techniques in Section II, there are many studies on integrating different testing and/or debugging techniques.

For instance, Wong and colleagues proposed an approach to combining test suite minimization and prioritization to select cases based on the cost per additional coverage [22][23]. Baudry et al. [3] used a bacteriologic approach to generate test suites that aim at maximizing the number of dynamic basic blocks to make the fault localization more effective. Yu and colleagues examined the effect of test suite reduction on fault localization [24]. Their studies found that test suite reduction does have an impact on the effectiveness of fault localization techniques. However, they neither studied test case prioritizations nor the extent of reductions that may lead to effective fault localizations similar to what we report in this paper.

Jiang et al. [13] examined the integration of test case prioritization and fault localization. They found that test case prioritization has an impact on the effectiveness of fault localization techniques and many existing prioritization techniques are no better than random ordering. However, they did not study to what extent test case prioritizations may generate test suites that existing fault localization techniques may use to locate faults effectively. Gonzalez-Sanchez et al. [9] proposed a new test case prioritization approach that maximizes the improvement of the diagnostic information per test. Their results showed that their technique could reduce the overall testing and debugging cost in some scenarios. They did not examine the effect of adequate test suites on fault localization techniques, however.

There are abundant studies on test case prioritization techniques. Srivastava and Thiagarajan [21] developed a binary matching technique to compute the changes in programs at the basic block level and prioritize test cases to cover maximally the affected program changes. Li et al. [16] evaluated various search algorithms for test cases prioritization. Leon et al. [15] also proposed failure-pursuit sampling techniques. Their failure-pursuit sampling uses one-per-cluster sampling to select the initial sample and, if a failure is found, its k nearest neighbors are selected and checked. If additional failures are found, the process will be repeated.

There are also studies on fault localization techniques that are closely related to the four techniques used in our experiment. For instance, Cleve and Zeller [5] proposed delta debugging, which automatically isolates failure-inducing inputs, produces cause-effect chains, and finds the faults. Renieris and Reiss [19] found that the use of the execution trace difference between a failed run and its nearest passed neighbor run is more effective than using other pairs for fault localization. Jeffrey et al. proposed a value-profile based approach to ranking program statements according to their likelihood of being faulty [10]. Zhang et al. [26] proposed to differentiate the short-circuit evaluations of individual predicates in individual program statements and produce one set of evaluation sequences per predicate for fault localization. Zhang et al. [25] proposed to use a network propagation approach to address the issue of coincidental correctness that may occur in test executions.

Because our study is an integration of test case prioritization techniques and fault localization techniques, the experiment will grow tremendously when we evaluated more fault localization techniques. We have, therefore, focused this work on the four most typical fault localization techniques in our study to make the empirical study manageable without losing representativeness.

V. Conclusion

Program debugging can be initiated even though testing only produces partial test results. This leads to the problem of selecting and executing some test cases before executing others, and whether the executed test cases with test results can effectively help debugging.

Fault localization is one of the major tasks in debugging. Our work has shown that existing and practical test suite adequacy criteria are still insufficient in the effective support of statistical fault localization techniques.

It appears to us that the notion of using adequacy as a criterion to stop the testing effort may have adverse effects on fault localization effectiveness. We have also found that branch-adequate test suites are significantly better than statement-adequate test suites in effectively supporting fault localization. Furthermore, we have conducted an analysis of existing fault localization techniques and have found that they still ineffectively rank faulty statements within a (small) debugging panel in typical IDEs. When branch-adequate test suites are used by existing test case prioritization techniques to identify higher priority test cases with the aim of supporting effective fault localization, we have found that the saving is not impressive. The results have shown that there are still large gaps in integrating various kinds of testing and debugging techniques so that they can be used by developers under one roof.

Test adequacy criteria are important because they define when to stop testing amid an infinite number of possible test cases in the input domain. Random testing can be effectively used, say, to crash test a program. We believe that random testing and adequacy testing are useful for different purposes. As a result, our paper focuses on the impact of adequate test suites on fault localization. It will be interesting to study other useful and practical testing techniques and resolve their effective integration. It will also be interesting to study reliability testing and its integration with program debugging.

VI. REFERENCE

[1] The economic impacts of inadequate infrastructure for software testing. Final Report. National Institute of Standards and Technology, Gaithersburg, MD, 2002. Available at http://www.mel.nist.gov/msid/sima/sw_testing_rpt.pdf.

[2] R. Abreu, P. Zoeteweij, and A. J. C. van Gemund. On the accuracy of spectrum-based fault localization. In *Proceedings of the Testing: Academic and Industrial Conference: Practice And Research Techniques* (*TAICPART-MUTATION 2007*), pages 89–98. IEEE Computer Society Press, Los Alamitos, CA, 2007.

[3] B. Baudry, F. Fleurey, and Y. Le Traon. Improving test suites for efficient fault localization. In *Proceedings of the 28th International Conference on Software Engineering* (*ICSE 2006*), pages 82–91. ACM Press, New York, NY, 2006.

[4] C. Cadar, D. Dunbar, and D. R. Engler. KLEE: unassisted and automatic generation of high-coverage tests for complex systems programs. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation* (*OSDI 2008*), pages 209–224. USENIX Association, Berkeley, CA, 2008.

[5] H. Cleve and A. Zeller. Locating causes of program failures. In *Proceedings of the 27th International Conference on Software Engineering* (*ICSE 2005*), pages 342–351. ACM Press, New York, NY, 2005.

[6] H. Do, S. G. Elbaum, and G. Rothermel. Supporting controlled experimentation with testing techniques: an infrastructure and its potential impact. *Empirical Software Engineering*, 10 (4): 405–435, 2005.

[7] S. G. Elbaum, A. G. Malishevsky, and G. Rothermel. Test case prioritization: a family of empirical studies. *IEEE Transactions on Software Engineering*, 28 (2): 159–182, 2002.

[8] S. G. Elbaum, G. Rothermel, S. Kanduri, and A. G. Malishevsky. Selecting a cost-effective test case prioritization technique. *Software Quality Control*, 12 (3): 185–210, 2004.

[9] A. Gonzalez-Sanchez, E. Piel, H.-G. Gross, and A. J. C. van Gemund. Prioritizing tests for software fault localization. In *Proceedings of the 10th International Conference on Quality Software* (*QSIC 2010*), pages 42–51. IEEE Computer Society Press, Los Alamitos, CA, 2010.

[10] D. Jeffrey, N. Gupta, and R. Gupta. Fault localization using value replacement. In *Proceedings of the 2008 ACM SIGSOFT International Symposium on Software Testing and Analysis* (*ISSTA 2008*), pages 167–178. ACM Press, New York, NY, 2008.

[11] B. Jiang and W. K. Chan. On the integration of test adequacy: test case prioritization and statistical fault localization. *The 1st International Workshop on Program Debugging in China* (*IWPDC 2010*), in Proceedings of the 10th International Conference on Quality Software (*QSIC 2010*), IEEE Computer Society Press, Los Alamitos, CA, pages 377–384, 2010.

[12] B. Jiang, Z. Zhang, W. K. Chan, and T. H. Tse. Adaptive random test case prioritization. In *Proceedings of the 24th IEEE/ACM International Conference on Automated Software Engineering* (*ASE 2009*), pages 233–244. IEEE Computer Society Press, Los Alamitos, CA, 2009.

[13] B. Jiang, Z. Zhang, T. H. Tse, and T. Y. Chen. How well do test case prioritization techniques support statistical fault localization. In *Proceedings of the 33rd Annual International Computer Software and Applications Conference* (*COMPSAC 2009*), volume 1, pages 99–106. IEEE Computer Society Press, Los Alamitos, CA, 2009.

[14] J. A. Jones, M. J. Harrold, and J. Stasko. Visualization of test information to assist fault localization. In *Proceedings of the 24th International Conference on Software Engineering* (*ICSE 2002*), pages 467–477. ACM Press, New York, NY, 2002.

[15] D. Leon, W. Masri, and A. Podgurski. An empirical evaluation of test case filtering techniques based on exercising complex information flows. In *Proceedings of the 27th International Conference on Software Engineering* (*ICSE 2005*), pages 412–421. ACM Press, New York, NY, 2005.

[16] Z. Li, M. Harman, and R. M. Hierons. Search algorithms for regression test case prioritization. *IEEE Transactions on Software Engineering*, 33 (4): 225–237, 2007.

[17] B. Liblit, M. Naik, A. X. Zheng, A. Aiken, and M. I. Jordan. Scalable statistical bug isolation. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation* (*PLDI 2005*), ACM SIGPLAN Notices, 40 (6): 15–26, 2005.

[18] C. Liu, X. Yan, L. Fei, J. Han, and S. P. Midkiff. SOBER: statistical model-based bug localization. In *Proceedings of the Joint 10th European Software Engineering Conference and 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering* (*ESEC 2005/FSE-13*), ACM SIGSOFT Software Engineering Notes, 30 (5): 286–295, 2005.

[19] M. Renieris and S. P. Reiss. Fault localization with nearest neighbor queries. In *Proceedings of the 18th IEEE International Conference on Automated Software Engineering* (*ASE 2003*), pages 30–39. IEEE Computer Society Press, Los Alamitos, CA, 2003.

[20] G. Rothermel, R. H. Untch, C. Chu, and M. J. Harrold. Prioritizing test cases for regression testing. *IEEE Transactions on Software Engineering*, 27 (10): 929–948, 2001.

[21] A. Srivastava and J. Thiagarajan. Effectively prioritizing tests in development environment. In *Proceedings of the 2002 ACM SIGSOFT International Symposium on Software Testing and Analysis* (*ISSTA 2002*), ACM SIGSOFT Software Engineering Notes, 27 (4): 97–106, 2002.

[22] W. E. Wong, V. Debroy, and B. Choi. A family of code coverage-based heuristics for effective fault localization. *Journal of Systems and Software*, 83 (2): 188–208, 2010.

[23] W. E. Wong, J. R. Horgan, S. London, and H. Agrawal. A study of effective regression testing in practice. In *Proceedings of the 8th*

International Symposium on Software Reliability Engineering (*ISSRE 1997*), pages 264–274. IEEE Computer Society Press, Los Alamitos, CA, 1997.

[24] Y. Yu, J. A. Jones, and M. J. Harrold. An empirical study of the effects of test-suite reduction on fault localization. In *Proceedings of the 30th International Conference on Software Engineering* (*ICSE 2008*), pages 201–210. ACM Press, New York, NY, 2008.

[25] Z. Zhang, W. K. Chan, T. H. Tse, B. Jiang, and X. Wang. Capturing propagation of infected program states. In *Proceedings of the 7th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT International Symposium on Foundations of Software Engineering* (*ESEC 2009/FSE-17*), pages 43–52. ACM Press, New York, NY, 2009.

[26] Z. Zhang, B. Jiang, W. K. Chan, T. H. Tse, and X. Wang. Fault localization through evaluation sequences. *Journal of Systems and Software*, 83 (2): 174–187, 2010.