HKU CS Tech Report TR-2008-07

Postprint of article in *Proceedings of the 32nd Annual International Computer Software and Applications Conference* (*COMPSAC '08*), IEEE Computer Society, Los Alamitos, CA, pp. 128–135 (2008)

Debugging through Evaluation Sequences: A Controlled Experimental Study * †

Zhenyu Zhang, Bo Jiang
The University of Hong Kong
Pokfulam, Hong Kong
{zyzhang, bjiang}@cs.hku.hk

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

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

Abstract

Predicate-based statistical fault-localization techniques locate fault-relevant predicates in a program by contrasting the statistics of the values of individual predicates between successful and failure-causing runs. While short-circuit evaluations are common in program execution, treating predicates as atomic units ignores this fact, masking out various types of important statistics. On the contrary, are such statistics useful for debugging? In this paper, we investigate experimentally the impact of the use of short-circuit evaluation information on fault localization. The results show that, by doing so, it significantly improves predicate-based statistical fault-localization techniques.

Keywords: evaluation sequence, fault localization.

1 Introduction

Software debugging is a key activity in software development, and takes up a significant amount of resources in a typical project. Among the three major tasks of software debugging (namely, fault localization, fault repair, and regression testing of repaired programs), fault localization has been recognized as the hardest, tedious, and time-consuming [14]. Using an effective fault-localization technique to improve the productivity of programmers is a long-standing trend to alleviate the problem.

Recently, effective statistical fault-localization techniques were proposed. A strategy [10, 11] is to identify fault-relevant predicates rather than directly pinpointing the fault locations. This strategy holds the promise to sample a program and collect execution statistics in a lightweight manner, which also reduces the need to disclose the execution details of all statements when remote sampling is conducted (for the purpose of remote support rather than on-site support). Hence, it lowers the risk of information leakage, which is a security concern.

These techniques, however, need to summarize the execution statistics on individual predicates. A compound predicate may be executed in one way or another due to short-circuit evaluations over different sub-terms of the predicate. The execution statistics of a predicate is, therefore, the summary of a collection of lower-tier evaluations over different sub-terms. *Is isolating these lower-tier evaluations beneficial in improving the effectiveness of predicate-based statistical fault-localization techniques?* This paper conducts a controlled experimental investigation on the impact of the use of short-circuit evaluation sequences to improve statistical fault localization techniques.

We first give a few preliminaries. A *successful test case* is a test case showing no failures, and a *failure-causing test case* is one that detects a failure. A typical program contains numerous *predicates* in if- and while-statements. They are in the form of *Boolean expressions*, such as "`*j<=1 || src[*i+1]=='\0'`", which may comprise further *conditions*, such as "`*j<=1`" and "`src[*i+1]=='\0'`".

Previous studies on statistical fault localization [10, 11] find the fault-relevant predicates in a program by counting the number of times (n_t) a predicate is evaluated to be true in an execution as well as the number of times (n_f) it is evaluated to be false, and then comparing these counts in

---

† This project is supported in part by the General Research Fund of the Research Grants Council of Hong Kong (project nos. 111107 and 716507).

‡ All correspondence should be addressed to Prof. T.H. Tse at Department of Computer Science, The University of Hong Kong, Pokfulam, Hong Kong. Tel: (+852) 2859 2183. Fax: (+852) 2858 4141. Email: thtse@cs.hku.hk.

various ways. The *evaluation bias* $\frac{n_t}{n_t+n_f}$ of a predicate is the percentage that it is evaluated to be true among all evaluations in a run [11].

The SOBER approach [11] proposes to contrast the differences between a set of evaluation biases due to successful test cases and that due to failure-causing ones for every predicate in the program. It hypothesizes that, the greater is the difference between such a pair of sets of evaluation biases, the higher will be the chance that the corresponding predicate is fault-relevant. The CBI approach [10] proposes a heuristic that measures the increase in probability that a predicate is evaluated to be true in a set of failure-causing test cases, compared to the whole set of (successful and failure-causing) test cases. These proposals are particularly interested in the evaluation results of predicates. They use the resultant values of the predicates to determine the counts.

A predicate can be considered as a Boolean expression. As mentioned above and to be discussed in Section 2, the resultant values of a Boolean expression may be due to different evaluation sequences. If we ignore the information on evaluation sequences, we may be masking out very useful statistics for effective fault localization. In this paper, we investigate whether the effect of a lower-tier concept — evaluation sequences — of predicates is significant on the effectiveness of predicate-based statistical fault localization. We set up a controlled experiment to study this question.

The major contributions of this paper are twofold: (*i*) We provide the first set of experimental results regarding the effect of short-circuit evaluations on statistical debugging. (*ii*) We show that short-circuit evaluation has a significant impact on the effectiveness of predicate-based fault-localization techniques. Indeed, the experimental result shows that the use of evaluation sequences can significantly improve on existing predicate-based statistical fault-localization techniques.

We shall illustrate the potential of using evaluation sequences for fine-grained statistical fault localization in Section 2, which casts a scene for us to formulate the research questions in Section 3, followed by the associated experiment in Section 4. We shall next review related work in Section 5. Section 6 concludes the paper.

2 A Motivating Study

This section shows a motivating study we have conducted. It enables readers to have a feel of how the distribution of evaluation biases at the evaluation sequence level can be used to pinpoint a faulty predicate.

The upper part of Figure 1 shows a code fragment excerpted from the original version (version v0) of print_tokens2 from the Siemens suite of programs [5]. We have labeled the three individual conditions as C_1, C_2, and C_3, respectively. The lower part of the same figure shows the code fragment excerpted from a faulty version (version v8) of the Siemens suite, where a fault was seeded into the predicate by adding an extra condition `ch=='\t'`. We have labeled this condition as C_4.

`/* Original Version v0 */`

$$\texttt{if}\,(\overbrace{ch == \texttt{' '}}^{C_1}\,||\,\overbrace{ch == \texttt{'\textbackslash n'}}^{C_2}\,||\,\overbrace{ch == 59}^{C_3})$$

`    return(true);`

`/* Faulty Version v8 */`

$$\texttt{if}\,(\overbrace{ch == \texttt{' '}}^{C_1}\,||\,\overbrace{ch == \texttt{'\textbackslash n'}}^{C_2}\,||\,\overbrace{ch == 59}^{C_3}\,||\,\overbrace{ch == \texttt{'t'}}^{C_4})$$

`    return(true);`

Figure 1. Code excerpts from versions v0 and v8 of print_tokens.

ES	C_1	C_2	C_3	C_4	v0	v8	v0 = v8?
es_1	T	$\perp$	$\perp$	$\perp$	T	T	yes
es_2	F	T	$\perp$	$\perp$	T	T	yes
es_3	F	F	T	$\perp$	T	T	yes
es_4	F	F	F	T	F	T	**no**
es_5	F	F	F	F		F	yes

Table 1. Evaluation sequences of code fragments.

Because of the effect of short-circuit rules of the C programming language on Boolean expressions, a condition in a Boolean expression may be evaluated to be `true` (`T`) or `false` (`F`), or may not be evaluated at all ($\perp$). Furthermore, in terms of evaluations, the conditions on a Boolean expression can be seen as an ordered sequence. [1] When a preceding condition in an evaluation sequence is not evaluated, by the short-circuit rule, no succeeding condition in the evaluation sequence will be evaluated.

For the faulty Boolean expression in the fragment shown in Figure 1, there are five legitimate evaluation sequences (es_1 to es_5), as shown in Table 1. The columns under the individual conditions (C_1 to C_4) represent the evaluation outcomes of the respective conditions based on the short-circuit rules of the programming language. In the column entitled v0, it shows the respective resultant values of the predicate in the original version of the program. In this column, the last two grids are merged because the two evaluation sequences (es_4 and es_5) make no difference in

[1] We simply consider every condition to be a distinct occurrence. In other words, even if two conditions in a predicate are identical, we consider them as two distinct occurrences.

the original program. The column entitled v8 shows the respective resultant values in the faulty program. The rightmost column shows whether the original and faulty predicates give the same values.

To gain an idea of whether short-circuit rules can be useful for fault localization, we have run an initial experiment. We apply the whole test pool for the program from the Software-artifact Infrastructure Repository (SIR) [5], and record the counts of each of the five evaluation sequences for each test case. Following [11], we use the formula in Section 1 to calculate the evaluation biases for the set of successful test cases, and those for the set of failure-causing test cases. The results are shown as the histograms in Figure 2. The distribution of evaluation biases over successful test cases and that over failure-causing test cases are given in pairs. The plots in Figures 2(a) to 2(e) are the respective distribution pairs of the five evaluation sequences. The plots in Figures 2(f) and 2(g) are those for the predicate-level, as used in previous work ([11]).

From the histograms in Figure 2, we observe that the distribution of evaluation biases for es_4 on successful test cases is drastically different from that of the failure-causing one. Indeed, it is the most different one among all pairs of histograms shown in the figure. We also observe from Table 1 that the fault in the code fragment can only be revealed when es_4 is used, because the fault does not affect the values in the other alternatives.

Our initial study indicates that it may be feasible to use evaluation sequences to identify a fault-relevant statement more accurately. However, it is still uncertain how much the use of evaluation sequences will be beneficial to fault localization. We shall formulate our research questions in the next section and then investigate them experimentally in Section 4.

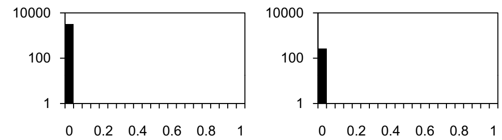
3 Research Questions

In this section, we shall discuss the research questions to be addressed by our controlled experimental study. We refer to a predicate-based statistical fault-localization technique as a *base* technique, and refer to the use of evaluation sequences in predicate execution counts as the *fine-grained* version of the base technique.

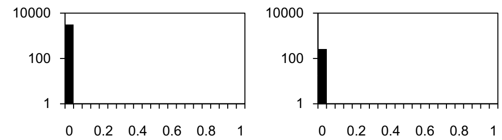
RQ1: In relation to the base technique, is the use of evaluation sequences for statistical fault localization effective?

RQ2: If the answer to **RQ1** is true, is the effectiveness of using evaluation sequences significantly better than the base technique?

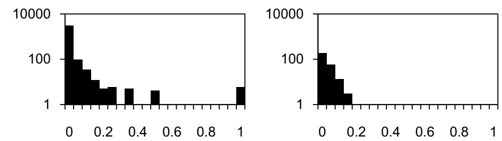
RQ3: Do the execution statistics of different evaluation sequences of the same predicate differ significantly?



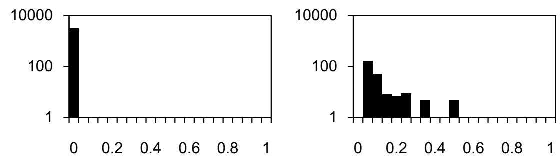
(a) es_1 Plot



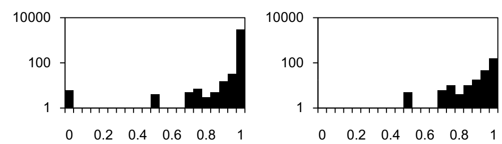
(b) es_2 Plot



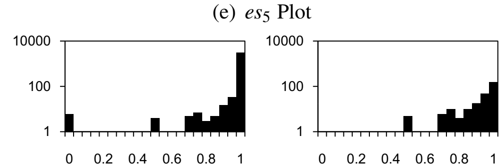
(c) es_3 Plot



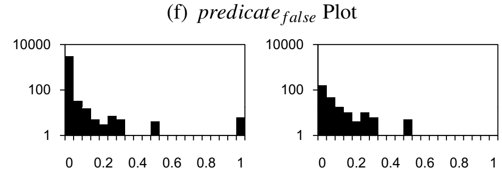
(d) es_4 Plot



(e) es_5 Plot



(f) $predicate_{false}$ Plot



(g) $predicate_{true}$ Plot

Figure 2. Comparison of distributions of evaluation biases (x-axis: evaluation bias; y-axis: no. of test cases).

3.1 Performance Evaluation

Performance metrics are widely used to facilitate comparisons among different approaches. Renieres and Reiss [13] propose a (T-score) method of for measuring their fault-localization technique. The method is also adopted by Cleve and Zeller [3] and Liu et al. [11] to evaluate other fault-localization techniques.

For the ease of comparison with previous work, we also use T-scores to evaluate the fine-grained evaluation sequence approach in relation to the corresponding base techniques. We select two base techniques for study, namely SOBER [11] and CBI [10], because they are representative.

In brief, the T-score method takes a program P, its marked faulty statements S, and a sequence of most suspected faulty statements S' as inputs, and produces a value V as output. The procedure to compute the T-score is as follows: (i) Generate a Program Dependence Graph (PDG) G for P. (ii) Using the dependence relations in the PDG as a measure of distance among statements, do a breadth-first search starting with the statements in S', until some statement in S is reached. (iii) Return the percentage of searched statements (with respect to the total number of statements in P) as the value V. If the original S' consists of k most suspected faulty statements, the final result is known as the top-k T-score value.

This measure is useful in assessing objectively the quality of proposed ranking lists of fault-relevant predicates and the performance of fault-localization techniques. Since the evaluation sequence approach is built on top of base techniques (such as SOBER and CBI), we also use T-scores to compare different approaches in our controlled experiment to answer the research questions.

3.2 Enabling Fine-Grained View of Base Techniques

As we are interested in studying the impact of short-circuit evaluations and evaluation sequences for statistical fault localization, we need a method to incorporate the fine-grained view into a base technique. Intuitively, this will provide execution statistics which may help statistical fault-localization techniques identify the locations of faults more accurately.

We note that a base technique, such as SOBER or CBI, conducts sampling of the predicates in a subject program to collect run-time execution statistics, ranks the fault relevance of the predicates. To assess the effectiveness of the selected set of predicates to locate faults, researchers may use T-scores to determine the percentage of code examined in order to discover the fault.

As such, given a set of predicates applicable to a base technique, we identify all legitimate evaluation sequences for each of these predicates. We then insert probes at the predicate locations to collect the evaluation outcomes of atomic conditions in these predicates. For each evaluation of a predicate, based on the evaluation outcomes of the atomic conditions, we can determine the evaluation sequence that takes place in the predicate evaluation. Hence, we collect the counts for individual evaluation sequences. By treating each evaluation sequence as a distinct (fine-grained) predicate in the base technique, the ranking approach in the base technique can be adopted to rank these fine-grained predicates.

On the other hand, from the developers' viewpoint, it may be more convenient to recognize (through their eyeballs) the occurrence of an original predicate (than an evaluation sequence of the predicate) from the program text. Hence, it is to the benefit of developers to map the ranked evaluation sequences to their respective predicates and thus the corresponding statements.

Some measures need to be taken in the above mapping procedure, however. Different evaluation sequences may receive different ranks. A simple mapping may thus result in a situation where a predicate occurs more than once in a ranking list. We choose to use the highest rank of all evaluation sequences for each individual predicate as the final rank of that predicate. This strategy also aligns with the basic idea of predicate ranking in SOBER and CBI. We refer to the fine-grained approach as *Debugging through Evaluation Sequences* (*DES*).

4 Controlled Experiment

This section presents a controlled experiment and its results and analyses.

4.1 Subject Programs and Test Cases

In this study, we choose the Siemens suite of programs to conduct our experiment. They were originally created to support research on data-flow and control-flow test adequacy [7]. Our version of Siemens subject programs are obtained from the Software-artifact Infrastructure Repository (SIR) [5] at http://sir.unl.edu. The Siemens suite consists of seven programs as shown in Table 2. A number of faulty versions are attached to each program. In our experiment, if any faulty version comes with no failure-causing cases, we do not include it in the experiment, since the base techniques [10, 11] require failure-causing test cases. We use a Unix tool, gcov, to collect the instrumentation log. Six faulty versions that cannot be processed by gcov are excluded. As a result, we use 126 faulty versions in total.

Each of the Siemens programs is equipped with a test pool. According to the authors' original intention, the test pool simulates a representative subset of the input domain of the program, so that test suites should be drawn from such a test pool [5]. In the experiment, we follow the work of [11] to input the whole test pool to every technique to rank predicates or their evaluation sequences.

Program	Exe. LoC	Faulty Ver.	A	B
print_tokens	341–342	7	4130	1.7
print_tokens2	350–354	10	4115	5.4
replace	508–515	31	5542	2.0
schedule	291–294	5	2650	3.2
schedule2	261–263	9	2710	1.0
tcas	133–137	41	1608	2.4
tot_info	272–274	23	1052	5.6

Exe. LoC: executable lines of code.
Faulty Ver.: no. of faulty versions.
A: no. of test cases in the test pool.
B: average percentage of compound Boolean expressions to all Boolean expressions.

Table 2. Statistics of subject programs.

Table 2 shows the statistics of the subject programs and test pools that we use. The data with respect to each subject program, including the executable lines of code (column "Exe. LoC"), the number of faulty versions (column "Faulty Ver."), the size of the test pool (column A), and the average percentage of compound Boolean expression statements with respect to all Boolean expression statements (column B), are obtained from SIR [5] (as at January 10, 2008). For instance, there are 10 faulty versions for the print_tokens2 program. Their sizes vary from 350 to 354 LoC, and their test pool contains 4115 test cases. On average, 5.4% of the Boolean expression statements in these faulty versions contain compound Boolean expressions. Other rows can be interpreted similarly.

We observe from column B that, in each subject program, the percentage of predicates having more than one atomic condition is low. This makes the research questions even more interesting: We would like to see whether such a low percentage would affect the performance of a base technique to a large extent.

4.2 Setup of Controlled Experiment

In this section, we describe the setup of the controlled experiment. Using our tool, we produce a set of instrumented versions of the subject programs, including both the original and faulty versions. Based on the instrumentation log as well as the coverage files created by gcov, we calculate the execution counts for the evaluation sequences, and finally rank the Boolean expression statements according to the description presented in Section 3. We also calculate the number of faults successfully identified through the examined percentage of code at different T-score values (see Section 3).

The experiment is carried out on a DELL PowerEdge 1950 server with two 4-core Xeon 5355 (2.66Hz) processors, 8GB physical memory and 400GB hard disk equipped, serving a Solaris Unix with the kernel version of Generic_120012-14.

Our experimental platform is constructed using the tools of flex++ 2.5.31, bison++ 1.21.9-1, CC 5.8, bash 3.00.16(1)-release (i386-pc-solaris2.10), and sloccount 2.26.

4.3 Results and Analysis

In this section, we present the experimental results, compare the relative effectiveness of the integrated approach with the base approach, and address the research questions one by one.

Answering RQ1: Is DES effective? Figures 3 and 4 show the results of SOBER against SOBER enabled with DES, and those of CBI against CBI enabled with DES, respectively. To ease our discussion, we refer to CBI enabled with DES as DES_CBI, and SOBER enabled with DES as DES_SOBER.

The x-axis of each plot in these two figures shows the T-score values, which represents the percentage of statements of the respective faulty program version to be examined. The y-axis is the percentage of faults located within the given code examining range. According to [11], the use of the top 5 predicates in the ranked list will produce the best results for both SOBER and CBI. For a fair comparison with previous work, we also adopt the use of the top 5 predicates in the controlled experiment. In the remaining parts of the paper, therefore, we shall always compare the top-5 T-score values for DES_SOBER and DES_CBI against those for SOBER and CBI.

We observe from Figure 3 that DES_SOBER consistently achieves better average fault localization results (that is, more faults for the same percentage of examined code) than SOBER. For example, when checking 10% to 20% of the code, DES_SOBER can find at least 10 percent more faults than SOBER. As the percentage of examined code increases, however, the difference shrinks. This is understandable because, when an increasing amount of code has been examined, the difference between marginal increases of located faults will naturally be diminished. When all the faults are located or all the statements are examined, the two curves will attain the same percentage of located faults. We also observe from Figure 4 that DES_CBI also outperforms CBI.

However, the visual differences between the curves appear to be small. To gain a more detailed picture, we further compare the two base techniques with their DES-enabled versions from another point of view. Figures 5 and 6 show the relative comparison of DES_SOBER and SOBER as well as DES_CBI and CBI on the Siemens suite of programs.

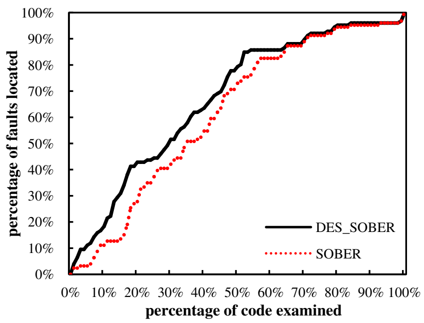


Figure 3. Direct comparison of DES_SOBER **and** SOBER**.**

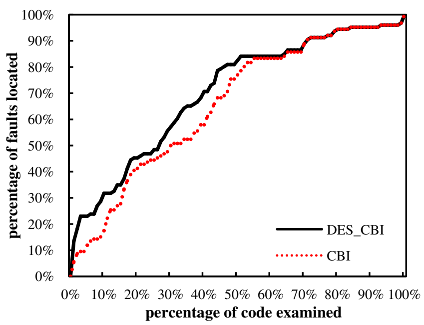


Figure 4. Direct comparison of DES_CBI **and** CBI**.**

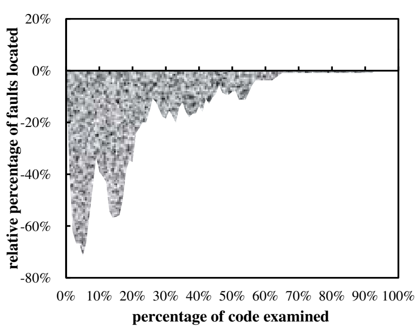


Figure 5. Relative comparison of DES_SOBER **and** SOBER**.**

In Figure 5, the x-axis of the plot is the percentage of code examined (T-score). For a given percentage of code examined, the y-axis shows a value that we call the *relative percentage of faults located*, calculated by the formula $\frac{\text{percentage of faults located by SOBER}}{\text{percentage of faults located by DES\_SOBER}} - 1$. Figure 6 can be interpreted similarly. In either figure, the parts below the x-axis indicates the relative percentage that the DES-enabled version of the respective technique (SOBER or CBI) outperforms the base version of the same technique. The portion above the x-axis shows the opposite.

Firstly, let us examine Figure 5. When the percentage

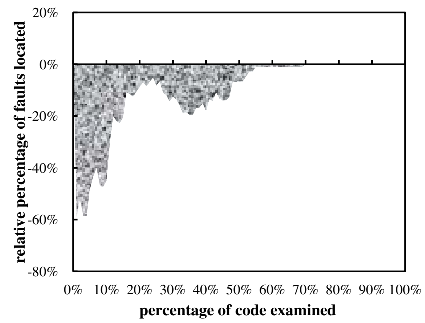


Figure 6. Relative comparison of DES_CBI **and** CBI**.**

of examined code is low, say 2% to 20%, the curves for SOBER is far below the x-axis. This shows that SOBER locates fewer faults than DES_SOBER when the percentage of code examined is small. Similarly, we can find from Figure 6 that CBI locates fewer faults compared with DES_CBI. When the T-score increases, the differences shrink as expected.

The results show that, on average, the evaluation sequence approach attains a relatively good fault-localization effectiveness (when benchmarked with the base techniques). We can, therefore, answer the first research question: the DES approach is effective.

Answering RQ2: Is DES better? In the above, we showed that the DES approach is effective for fault localization. However, it is unclear whether the difference between a base technique and its DES-enabled version is simply due to chance. We further wish to find out: Does a technique enabled with the evaluation sequence approach differ significantly from the base technique? Is it indeed better?

To answer these questions, we perform a Mann-Whitney U-test to determine whether a DES-enabled technique differs significantly from its base technique. The detailed procedure to analyze the data is as follows.

Firstly, we subtract the percentage of located faults within the given T-score (values of $0\%, 10\%, \ldots, 100\%$) of DES_CBI by that of CBI to obtain a set of sample data. Then, we compare this sample set with another set of data containing only zeros to test the null hypothesis that DES_CBI and CBI are not significantly different. The result of the U-test for DES_CBI and CBI gives a p-value [4] of less than 0.001, which successfully rejects the null hypothesis at 5% significant level. It confirms that DES_CBI and CBI are significantly different. We also perform a U-test using the same procedure on DES_SOBER and SOBER. It also gives a p-value of less than 0.001, which again successfully rejects the null hypothesis at 5%

significant level. From the experimentation above, we also observe that the DES-enabled versions improve on their base techniques.

Our answer to RQ2 is, therefore, that the DES-enabled techniques differ significantly from the respective base techniques. The answer to RQ2 also confirms that short-circuit evaluation rules do have significant impacts on statistical fault localization.

Combining the answers to RQ1 and RQ2, the experimental results show that the DES approach has the potential to improve significantly the effectiveness of fault-localization techniques. [2] They also show that short-circuiting is a significant factor in predicate-based statistical fault localization.

Answering RQ3: Do different evaluation sequences give the same result? To answer RQ3, we collect the execution statistics of all the evaluation sequences of the same Boolean expression to calculate the statistical differences between successful and failure-causing test cases. We perform a U-test between the evaluation biases for the sets of evaluation sequences over the same predicate in successful and failure-causing test cases. The results of the U-test shows that, for 59.12% of the evaluation sequences, there is a significant difference (at 5% significant level) between the evaluation biases of successful and failure-causing test cases. In other words, 59.12% of the evaluation sequences are useful fault location indicators, while the remaining 40.87% are not useful standalone fault predicators to differentiate failure-causing test cases from successful ones.

The answer to RQ3 is that different evaluation sequences of the same predicate may have different potentials for fault localization. It will be interesting to analyze the results further to know the reasons.

4.4 Threats to Validity

We briefly summarize below the threats to validity in our controlled experiment.

Construct validity is related to the platform dependence issues when using the Siemens programs in SIR [5]. Since every program in SIR has a fault matrix file to specify the test verdict of each test case (that is, whether a test case is successful or failure-causing), we also create a fault matrix file for our test results and carefully verify each test verdict against the corresponding one supplied by SIR. We observe that there are only minor differences in test verdicts between the two fault matrix files. We have thoroughly verified our setting, and believe that the difference is due to platform dependence issues.

[2] We are conservative about the conclusion because it is subject to external threats to validity to generalize the results.

Internal validity is related to the risk of having confounding factors that affects the observed results. Following [11], in the experiment, each technique uses all the applicable test cases to locate fault-relevant predicates in each program. The use of a test suite with a different size may give a different result [11]. Evaluations on the impact of different test suite sizes on our technique would be welcome. Another important factor is the correctness of our tools. Instead of adopting existing tools used in the literature, we have implemented our own tools in C++ for the purpose of efficiency. To avoid errors, we have adhered to the algorithms in the literature and implemented and tested our tools carefully. To align with previous work, we use the T-score method to compute the results of this experiment. The use of other metrics may produce different results.

External validity is the degree to which the results can be generalized to test real-world systems. We use the Siemens suite in the experiment to verify the research questions because they are commonly used by researchers in testing and debugging studies with a view to comparing different work more easily. Further applications of our approach to more medium to large size real-life programs would strengthen the external validity of our work. Each of the faulty versions in our subject programs contains one fault. Despite the competent programmer hypothesis, real-life programs may contain more than one fault. Although Liu et al. have demonstrated in [12] that predicate-based techniques can be used to locate faults in programs that contain more than one fault, their effectiveness in this scenario is not well discussed. We shall address this threat in future work.

5 Related Work

There are rich categories of techniques in statistical fault localization. There are others besides the predicate-based category [10, 11].

Delta Debugging [3, 15] isolates failure-inducing input elements, produces cause-effect chains, and locates the faults through the analysis of program state changes during a failed execution against a passed one.

Jones et al. [9] propose a Tarantula approach to rank statements according to their relevance to program faults, which is estimated by a ratio between the percentages of failure-causing and successful test cases that execute the statement. They further use Tarantula to explore ways of classifying test cases to enable several test engineers to debug a faulty program in parallel [8].

Liblit et al. [10] propose a sparse sampling approach CBI to collect the statistics of predicates for statistical fault localization. They further adapt CBI to exploit the execution statistics of compound Boolean expressions constructed from program predicates to facilitate statistical debugging [1].

Renieres and Reiss [13] find the difference in execution traces between a failed execution and its "nearest neighbor" passed execution to be effective for debugging. Statements with unsymmetrical differences between failed and passed runs are regarded as faulty statements.

Baudry et al. [2] define a dynamic basic block as the set of statements executed by the same test cases in a test suite. They use a bacteriologic approach to remove test cases while maximizing the number of dynamic basic blocks, and use the algorithm in [9] to rank the statements. They manage to use fewer test cases than Tarantula for the same fault-localization results.

Griesmayer et al. [6] use model checking to locate faults. By searching the error traces, expressions that repair the original program can be constructed.

6 Conclusion

Program debugging is time-consuming but important in software development. A major task in debugging is to locate faults. A common approach in statistical fault localization aims at locating program predicates that are close to faulty statements. This relaxes the requirement to pinpoint a fault location and has been shown empirically to be quite effective.

Following this popular trend, we would like to explore a better way to measure and rank predicates with respect to fault relevance. We observe that the fault-localization capabilities of various evaluation sequences of the same Boolean expression are not identical. Because of short-circuit evaluations of Boolean expressions in program execution, different evaluation sequences of a predicate may produce different resultant values. This inspires us to investigate the effectiveness of using Boolean expressions at the evaluation sequence level for statistical fault localization. The experiment on the Siemens suite of programs shows that our approach is promising. Our future work will include locating faults in multi-fault programs using representative test suites.

References

[1] P. Arumuga Nainar, T. Chen, J. Rosin, and B. Liblit. Statistical debugging using compound Boolean predicates. In *Proceedings of ISSTA '07*, pages 5–15. ACM, 2007.

[2] B. Baudry, F. Fleurey, and Y. Le Traon. Improving test suites for efficient fault localization. In *Proceedings of ICSE '06*. ACM, 2006.

[3] H. Cleve and A. Zeller. Locating causes of program failures. In *Proceedings of ICSE '05*, pages 342–351. ACM, 2005.

[4] G.E. Dallal. Why P = 0.05? 2007. Available at http://www.tufts.edu/ gdallal/p05.htm.

[5] 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.

[6] A. Griesmayer, S. Staber, and R. Bloem. Automated fault localization for C programs. *Electronic Notes in Theoretical Computer Science*, 174 (4): 95–111, 2007.

[7] M. Hutchins, H. Foster, T. Goradia, and T. Ostrand. Experiments on the effectiveness of dataflow- and controlflow-based test adequacy criteria. In *Proceedings of ICSE '94*, pages 191–200. IEEE Computer Society, 1994.

[8] J.A. Jones, M.J. Harrold, and J.F. Bowring. Debugging in parallel. In *Proceedings of ISSTA '07*, pages 16–26. ACM, 2007.

[9] J.A. Jones, M.J. Harrold, and J. Stasko. Visualization of test information to assist fault localization. In *Proceedings of ICSE '02*, pages 467–477. ACM, 2002.

[10] B. Liblit, M. Naik, A.X. Zheng, A. Aiken, and M.I. Jordan. Scalable statistical bug isolation. *in Proceedings of PLDI '05*, *ACM SIGPLAN Notices*, 40 (6): 15–26, 2005.

[11] C. Liu, L. Fei, X. Yan, S.P. Midkiff, and J. Han. Statistical debugging: a hypothesis testing-based approach. *IEEE Transactions on Software Engineering*, 32 (10): 831–848, 2006.

[12] C. Liu, X. Yan, L. Fei, J. Han, and S.P. Midkiff. SOBER: statistical model-based bug localization. In *Proceedings of ESEC 2005/FSE-13*, *ACM SIGSOFT Software Engineering Notes*, 30 (5): 286–295, 2005.

[13] M. Renieres and S.P. Reiss. Fault localization with nearest neighbor queries. In *Proceedings of ASE '03*, pages 30–39. IEEE Computer Society, 2003.

[14] I. Vessey. Expertise in debugging computer programs: a process analysis. *International Journal of Man-Machine Studies*, 23 (5): 459–494, 1985.

[15] A. Zeller and R. Hildebrandt. Simplifying and isolating failure-inducing input. *IEEE Transactions on Software Engineering*, 28 (2): 183–200, 2002.

[16] A.X. Zheng, M.I. Jordan, B. Liblit, M. Naik, and A. Aiken. Statistical debugging: simultaneous identification of multiple bugs. In *Proceedings of ICML '06*, pages 1105–1112. ACM, 2006.