2018 IEEE International Conference on Software Quality, Reliability and Security Companion

MURE: Making Use of MUtations to REfine Spectrum-Based Fault Localization

Zijie Li
State Key Laboratory of Computer Science
Institute of Software, Chinese Academy of Sciences
Beijing, China
lizj2018@ios.ac.cn

Lanfei Yan
State Key Laboratory of Computer Science
Institute of Software, Chinese Academy of Sciences
Beijing, China
yanlf@ios.ac.cn

Yuzhen Liu, Zhenyu Zhang*
State Key Laboratory of Computer Science
Institute of Software, Chinese Academy of Sciences
Beijing, China
{liuyz, zhangzy}@ios.ac.cn

Bo Jiang
School of Computer Science and Engineering
Beihang University
Beijing, China
jiangbo@buaa.edu.cn

***Abstract*—Locating faults in programs is never an easy task. Spectrum-based fault localization (SBFL) techniques estimate suspicious statements by contrasting the coverage spectra collected from passed and failed program runs. Mutation-based such techniques locate faults by trying different mutates with the aim of finding one that involves less turbulence to program behavior. The latter is empirically known more accurate, but with massive increases in time complexity. In this paper, we propose a new approach, MURE, which uses methodology of the latter to refine results of the former. MURE first drives a state-or-the-art SBFL technique Naish2 to output a list of suspicious statements. It then picks out suspicious statement as candidates, generates mutates for them, and estimates their likelihood of relating to faults. Finally, it refines the resultant list by adjusting part of its ordering. An experiment validates its effectiveness by showing a 30% accuracy improvement over Naish2.**

Keywords-software debugging; fault localization; mutation testing

I. INTRODUCTION

Programs, as a kind of man-made product, are often reported to contain errors. Software testing methods are used to validate the behaviors of programs. However, the fault localization task (after the confirmation of the existence of fault) is well known time-consuming. With the rapid growing of program scales and complexities, conventional approaches like setting breakpoints or step tracing can be neither effective nor efficient.

Fault localization is always one of the most time-consuming tasks in software debugging. In practice, even if a program contains a fault, there may exist program runs that reveal no failure. *Coincidental correctness*, which occurs when no failure is observed even though a fault has been triggered, is one of the reasons [22]. It makes identification of the fault even harder. Spectrum-Based Fault Localization (**SBFL**) has been proposed as an automatic mechanism to locate faults in programs [1][2][8][12][34]. The basic conjecture of SBFL is that a program statement is more suspicious (or safer) if it is exercised by more failed (or passed) test cases [14][27]. A large number of researchers put the main idea of SBFL to collect two types of execution information, namely, passed and failed, and then generate a ranked list of suspicious program statements according to the suspiciousness values calculated for each of them by a SBFL formula. Common SBFL formulas include *Jaccard* [2], *Ochiai* [1], and *Tarantula* [14] to name a few. Xie et al. [32] conducted a theoretical analysis to compare the accuracies of 30 SBFL formulas, and concluded that *Naish2* gives optimal result under specific conditions. However, recent studies shows that coincidental correctness disrupts the basic conjecture of SBFL and can adversely impact the accuracies of all the SBFL formulas [5][26][18][19].

Without having to tell the coincidental correctness runs, *failure rate* [35], an index capturing how many percentage of program runs fail, is also used in fault localization. Mutation testing techniques study a program by introducing mutation fault into it [12]. Suppose a program contains a mutation fault. We deem the new program, which is synthesized by embedding a new mutant at the location of the existing mutant, yet another one-mutant faulty program. Moon et al. [20], which developed the approach of mutation-based fault localization (**MBFL**), foresaw such a program less faulty than a double-mutant faulty program, which is synthesized by embedding a second mutant (i.e., not at the location of the existing mutant). They iterated every program statement with the aim of finding a location, embedding a mutant where will not increase the failure rate too much. Papadakis and Le Traon [23] proposed a more generalized another technique to locate "unknown" faults. Empirical studies showed that MBFL techniques are more accurate than SBFL techniques in locating faults, while come with huge additional resource usages caused by the need to execute a great deal of mutation versions of the target program [20][23].

Realizing the differences between the two kinds of techniques, we wonder whether there can be an approach integrating them. The goal is challenging because we have neither priori knowledge that where the faults reside nor the way of identifying passed test cases where coincidental correctness has occurred. In this paper we propose a new model called MURE, which uses the methodology of the latter to refine the results of the former. MURE first manipulates Naish2 to generate a fault-localization result, which is in form of a list of suspicious program statements. It then picks out the most suspicious statements from the list to conduct a mutation generation. More specifically, MURE generates limited number of mutants at the location of each

* All correspondence should be addressed to Zhenyu Zhang at Institute of Software, Chinese Academy of Sciences. Tel: (+8610) 62661630. Fax: (+8610) 62661627. Email: zhangzy@ios.ac.cn.

978-1-5386-7839-8/18/$31.00 ©2018 IEEE
DOI 10.1109/QRS-C.2018.00024

such suspicious statement, and obtains a set of mutation versions for the target program accordingly. By running those mutation versions and comparing over which test cases will they reveal failure, MURE finally assess the likelihood each generated mutation is at a location close to the original fault. Finally, it re-sorts the evaluated suspicious statements and synthesizes a new ranked list. We analytically show that MURE is of a lower time complexity than representative MBFL approaches such as [20]. To further validate the effectiveness of our proposal, we conduct an experiment using seven programs form Siemens suite [11], which have been extensively used in previous studies [1][2][15][18][28]. Empirical results confirm the effectiveness of MURE by showing a 30% improvement in fault-localization accuracy over the original Naish2.

The contribution of this paper is as follows. (i) It presents a novel approach to refine the results of SBFL techniques. (ii) It proposes a new fault localization technique MURE based on Naish2. (iii) It shows that MURE can significantly improve the accuracy of Naish2, with controllable additional cost.

The rest of this paper is organized as follows. Section II introduces related work. Section III presents our model. Section IV and V report the experiment results and case study, respectively. Section VI concludes the paper.

II. Related work

Related work are listed out in three parts in this section.

A. Spectrum-Based Fault Localization (SBFL)

A program spectrum is a collection of data that provides a specific view on the dynamic behavior of software. Generally speaking, it records the run-time profiles about various program statements for a specific test suite. The program statements could be statements, branches, paths, basic blocks, etc.; while the run-time information could be the binary coverage status, the number of time that the statement has been covered, and the program state before and after executing the program statement, and so on.

A spectrum-based fault localization technique compares program execution spectra of the same statements in passed or failed execution. Given a program $P = \cdot S_1, S_2, S_3, \ldots, S_n \cdot$ that contains n statements and runs against a test suite of m test cases $TS = \{t_1, t_2, t_3, \ldots, t_m\}$. After running P against TS, we get the execution information, with respect to an statement S_i, reformed into a vector, denoted as $A_i = \cdot a^i_{ef}, a^i_{ep}, a^i_{nf}, a^i_{np} \cdot$, where a^i_{ef} and a^i_{ep} represent the number of failed test cases and that of passed test cases in TS that covered S_i, respectively, a^i_{nf} and a^i_{np} denote the numbers of the two kinds of test cases that do not cover S_i [32]. Here, whether a test case is passed (or failed) is determined according to the result information of the program run over that test case.

The main component of a SBFL technique is the ranking function (aka. a SBFL formula), which maps the statement coverage data A_i to a suspiciousness value, and a ranked list of statements, which is compiled in the descending order of the suspiciousness values. A statement with a higher suspiciousness value implies a higher possibility to be faulty. Existing SBFL techniques include Jaccard [2], Tarantula [14], Ochiai [1], and so on. Many researchers focus on the accuracies of a SBFL formulas rather than the effectiveness of SBFL techniques, which may involve in other boosting mechanisms like tie breaking [29][30][231]. In a theoretical study [32], Xie and colleagues analytically confirmed that Naish2 gives an optimal accuracy under specific conditions like the single-fault assumption.

In this paper, we propose to refine the results of SBFL with mutation analysis as mentioned later in this paper.

B. Impact Factors to SBFL

SBFL is based on a simple conjecture: program elements that are mainly executed by failed test runs are more likely to be faulty, while those that are mainly executed by passed test runs are less likely to be faulty. Following this idea, when there are many test cases, which test results (passed or failed) are not correctly labeled, the suspiciousness of program statements will not be accurately evaluated [33].

Coincidental correctness happens when a fault is triggered but the program run does not reveal failures. Previous studies showed that it could adversely affect the effectiveness of software testing techniques [7][13]. Recent experimental evidence shows that it is undesirable to SBFL techniques as well. For example, Jones and colleagues [14] noticed that Tarantula fails to highlight faulty statements in the initialization code or main program path (e.g., code in the "main" function of a C program). They suggested that coincidental correctness may be the culprit and called for further investigation. Such adverse cases were also reported in other studies [5][24]. Coincidental correctness can be very common in practice and it is not easy to figure out which case triggers coincidental correctness. Up to date, some works try to reduce the negative impact of coincidental correctness by removing the dirty test cases whose coverage is not related to test effectiveness. For example, Wang et al. [28] proposed an approach using context patterns to refine code coverage information. Their approach, however, assumes the developers know the likely fault types, which may be difficult to decide in practice. Li et al. [16] proposed a cluster-analysis to alleviate coincidental correctness by grouping similar behavior test cases and reconstructing the coverage matrix, the failed test cases clustered too centralized or too scattered would lead to poor results. In practice, we cannot know ahead the pattern of coincidental correctness, so it is rather difficult to improve SBFL by identifying the passed cases that are suspected to incur coincidental correctness.

Many researchers have put their efforts to improving the accuracy of SBFL formulas. The quality of test suite is a key factor to impact the accuracy of SBFL. Our previous works focus on reducing the impact of test suites imbalance between passed cases and failed cases [34]. The work proposed in this paper is expected to integrate with the above approaches.

C. Mutation-Based Fault Localization (MBFL)

Mutation analysis, first proposed by Hamlet [10] and Demilo et al. [7], is a fault-based testing technique used to measure the effectiveness of a test suite. A mutation operator is a change-seeding rule to generate a mutant from the original program. Mutant operator contains many categories,

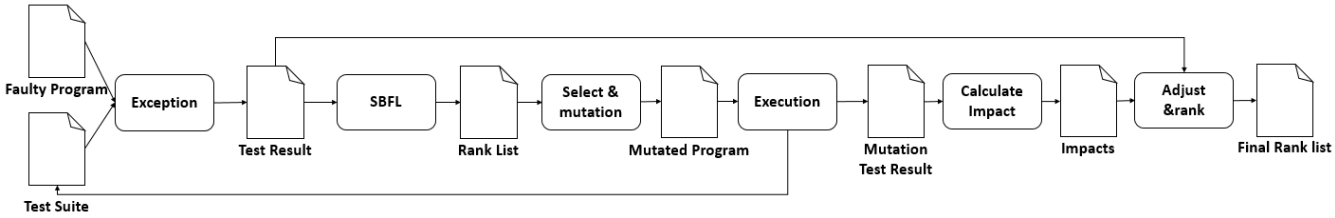

Figure 1. Overview of MURE

namely, statement mutations, operation mutations, variable mutations and constant mutations. Mutant operators in these categories are designed to model errors made by programmers in selection of identifiers and constants while formulating expressions, composition of expression functions and composition of functions using iterative and conditional statements [25].

Given the program P that runs against the test suite TS containing some failed test cases, which indicates that some program statements contain fault. Suppose S_f is the faulty statement, and S_c is a correct one. A mutation operation m is applied to statement S_i to generate a mutant P' for P. If the mutation is applied on S_f, we denote the mutated S_f as M_f, and similarly we denote the mutated S_c as M_c. Supposing that P contains only one fault, the former is a yet another one-mutant program while the latter contains an additional fault. Moon et al. [20] proposed a mutation-based fault localization approach based on a basic conjecture that a test case is more likely to fail on M_c than on M_f. Further, if P and P' differ only in one place which can be recovered by one mutation operation, we say P' is the first-order mutant [5]; otherwise a higher-order mutant. In this paper, we concern the former only.

Papadakis and Le Traon [23] iterated every program statement, generated mutation versions on each site, run them over the test suite, and compared the test results with the original test results to determine which mutation is embedded at a location close to the fault in the program. Empirical studies showed that their method achieves better accuracies than peer SBFL techniques. However, the additional cost is also significant since a MBFL technique often needs to execute a great deal of mutation versions of the target program.

Moon et al. [20] used mutation analysis called MUSE in fault localization, and the evaluation exhibited better effectiveness then SBFL. However, the execution of mutation analysis takes long time, and the fact that mutation operators needed vary among different programs constraints the utilization of the mutation analysis in fault localization. In order to solve this problem, we choose to combine mutant analysis with spectrum-based methods.

Peng et al. [10] used coverage information to choose total failed statement to inject mutants, which was designed to decrease the mutants generated, and then calculate the max value of mutants by SBFL to set the suspicious value of statement. But it did not consider the conversion between fail and pass, which is important on decreasing the impact of coincidental correctness in our analysis.

In our work, we discuss the possibility of using mutation-based methodology to refine the results of a SBFL technique.

III. Our Proposal - MURE

In this section we discuss the use of mutants to assist the fault localization process in details.

A. Framework

Figure 1 shows the framework of MURE, which consists of three major steps, namely, using SBFL to select promising statements for mutation, generating and testing the mutants, and the adjustment of the SBFL ranking list.

Step 1. We test a faulty program P with a test suite TS, and get the coverage information of each executed statement and the results of test cases. Then we generate a ranking list of suspiciousness values by utilizing a SBFL technique T. Further, we examine every statement in the ranking list, and select K most promising statements as candidates to be mutated.

Step 2. With respect to a mutation operator, a statement may yield many different mutants. To avoid explosive growth in size of mutants, we only randomly select N mutants per statement candidate. Then we test these mutation versions over the same test suite in Step 1, and collect their execution result.

Step 3. This step is to use an impact formula F to calculate an impact value for each statement candidate, and adjust the ranking list generated in Step 1.

Before we elaborate on each step, we first need to give necessary definitions to ease our presentation.

B. Definitions

Let $P(t_k)$ be the output of P over a test case t_k, and $P'(t_k)$ be the output of P' over the same test case t_j, where t_j belongs to TS.

As discussed, the relationship between $P(t_j)$ and $P'(t_j)$ can be one of the three cases. (i) **Equivalence.** It means $P(t_k)$ and $P'(t_k)$ are identical. In this case, the mutated program remains unchanged for test case t_k. For example, the mutated statement is not exercised, or the mutation generates a semantic equivalent change. (ii) **Passed to failed**. It implies that $P(t_k)$ yields a correct output, but $P'(t_j)$ results in a failed one. In particular, $P(t_k)$ is true but P' failed to produce the expected value on test case t_k. According to the basic conjecture, we are more likely to have mutated a correct statement. (iii) **Failed to passed**. In this case, $P(t_k)$ generates faulty output, while $P'(t_k)$ produces the correct one. In particular, $P(t_k)$ is false but P' produces the expected value on test case t_k. The case is not common in practice, which is

likely that the mutant is applied at the location of a faulty statement.

We thus define three terms to simply our presentation.

Definition 1: $M_{i,j}$ is the mutated statement S_i that adopts the j-th mutation on S_i. Let $Mc_{i,j}$ denotes the mutated statement while S_i is not faulty. Let $Mf_{i,j}$ denotes the mutated statement if S_i is faulty.

Definition 2: $Change_{pass \to fail}(M_{i,j})$ is the number of test cases, which test results change from passed to failed when running on the program that contains $M_{i,j}$.

Definition 3: $Change_{fail \to pass}(M_{i,j})$ is the number of test cases, which test results change from failed to passed when running on the program that contains $M_{i,j}$.

In the next section, we will elaborate on the implementation details of each step of our method MURE.

C. Implementation Details

1) The Selection of Candidates

Since our goal is to improve the accuracy of SBFL ranking, we do not necessarily mutate all statements in the program. Instead, we are only interested in those that might have ranked over the faulty statement in the SBFL ranking result. For example, consider a statement, which has never been executed by any failed test case. Mutating this statement is futile as it cannot be faulty, and CBFL techniques will generally rank it as the least suspicious statement.

Currently, our method selects only those statements that are covered by over K failed test cases to mutate, where K is a parameter that is decided by the user based on their estimation on the number of faults that have triggered the failed test cases. Suppose that all the failed test cases are caused by the same fault, as in our experiment. Then K is set to "Total Failed", which means only statements that covered by all failed cases are selected.

Finally, our approach is to select statements executed by all failed cases. One reason is that statements with more fault cases execution are more likely to be faulty. Another reason is due to the effectiveness of this model, applying this constraint could reduce the number of selected statements. It leads to a significant reduction of statements to be mutated, which reduces the time cost of our approach.

2) Mutant Sampling w.r.t. N:

According to the different syntactic elements in a statement, the mutation operator can be applied to the statement is different. Moreover, different mutation operator generates different number of mutants. In order to unify the number of mutants for each statement, we use mutant sampling technique, which was first proposed by Acree [4] and Budd [6]. A mutant sampling technique generates all possible mutants, and then selects randomly $x\%$ mutants to execute. Empirical studies of mutant sampling primarily focus on the random selection rate ($x\%$). Mathur and Wong [17] conducted an experiment to suggest that random selection of 10% of mutants is only 16% less effective than a full set of mutants in terms of mutation score.

In our approach, we randomly select mutants from mutants generated by mutating a specific statement, where the random selection rate is controlled to be 10% (i.e., $x\%$ = 10%). However, in this paper, we mainly devote to the effectiveness of our approach, rather than comparison between mutant reduction techniques. With such consideration, we choose mutant sampling to select 5 mutants to estimate the mutation impact for each statement.

But we found that some mutants would run into dead loop, or run too much long before the end. So we set a dynamic time-out for each case, if a case run out of time, we simply mark the case as failed. Unfortunately, some mutants may run out of time on all test suites, which take a long time for computer to finish testing the mutants. We call these mutants as Dead Mutants. To solve this problem, we set a threshold t so that if a mutant run out of time over $t\%$ of the whole test cases, we will drop it and randomly select from other m mutants correlated to the Dead Mutant.

3) The choice of base technique T and Impact Function F

Recent works by Naish and colleagues empirically compared existing ranking functions and find the best function they proposed is *Naish2* [21]. Our technique is based on this ranking function by selecting the T in Step 2 to be *Naish2*. So the enhancing formula of MURE can be equally expressed as

$$\mathrm{Susp}_{MURE} = Susp_{Naish2}(S_i) - Impact(S_i) \quad (1)$$

As presented above, we use subtraction to implement the refinement to the result of SBFL, and $Impact(S_i)$ in the formula is the value of our refinement, which will be discussed next. The suspiciousness value of *Naish2* is positive, and the higher the value, the more suspicious the statement is. And the impact value is the measurement how much the statement is likely to be correct, which has an adverse meaning with the suspicious value of Naish2, so the subtraction is selected.

The basic conjecture of MBFL means that it is more likely to occur that a passed test case will fail on mutation $Mc_{i,j}$ than a failed test case will pass on mutation $Mf_{i,j}$. So we define an impact value to map test suites changes to the likelihood of a statements' suspiciousness value.

$$\mathrm{Impact}(S_i) = \frac{\sum_{i=1}^{m} Change_{pass \to fail}(M_{i,j})}{m} - a\frac{\sum_{i=1}^{m} Change_{fail \to pass}(M_{i,j})}{m} \quad (2)$$

Here we add a coefficient a before the second term. The coefficient a balances the two terms, since it is harder to fix a fault by using mutation technique than to change a correct statement to false. In another word, the two terms contribute different weights to suspiciousness values. The coefficient a can be calculated as

$$a = \frac{|\, sum(pass \to fail) \,| + 1}{sum(pass)} \times \frac{sum(fail)}{|\, sum(fail \to pass) \,| + 1} \quad (3)$$

where $sum(pass \to fail)$ represents sum of $Change_{pass \to fail}(M_{i,j})$ for all mutants, $sum(fail \to pass)$ represents the sum of $Change_{fail \to pass}(M_{i,j})$ for all mutants, $sum(pass)$ denotes the number of executed passed cases before mutation, and $sum(fail)$ denotes the number of executed failed cases before mutation.

We say that when the value of impact is large, we are likely to mutate the correct statement. Similarly, when the

Table I. EXPERIMENT SUBJECTS

Programs	Number of Faulty Versions	Number of Test Cases	Percentage of Failed Test Cases	Number of Statements	Number of Executable Statements
tcas	41	1608	2.4%	138	63-67
schedule	9	2650	2.4%	299	151-154
schedule2	10	2710	3.2%	297	128-130
tot_info	23	1052	5.6%	346	122-123
print_tokens	5	4130	1.7%	402	194-195
print_tokens2	10	4115	5.4%	483	196-200
replace	31	5542	2.0%	516	241-246

value of impact is small, we are likely to mutate the faulty statement.

D. Complexity

Our technique attempts to exploit mutation analysis to enhance fault localization. However, we have to face a practical issue of mutation analysis: the high computational cost of executing enormous number of mutants against a test suite [12][22].

Program P contains n executable statements, while there are w statements executed by all failed cases; mutating each of them can yield l mutants; the corresponding test suite has m test cases. In our approach, our mutant sampling rate is $x\%$. We need to conduct $w \times m \times l \times x\%$ runs. Since the time complexity of SBFL is $O(n^2)$, and the time complexity is $O(n^2wmlx\%)$. Similarly, the time complexity of usual MBFL is $O(n^3ml)$. Obviously, w is smaller than n, and $x\%$ is smaller than 1. So MURE is efficient than the usual MBFL.

On the other hand, MURE is slower than SBFL. However, our goal of this work is to improve the accuracy of SBFL, as evaluated in the next section.

IV. Empirical Evaluation

A. Subject Program

We used seven C programs, *print_tokens*, *print_tokens2*, *replace*, *schedule*, *schedule2*, *tcas*, *tot_info* as subject programs. These programs are widely used in fault localization studies [3][15][31]. To determine whether a test case is passed or failed, we need to know the test oracle, so we run a fault-free version of each program.

Siemens suite contains a total of 129 versions of faulty programs. The information of subject programs in our experiment is shown in Table I. Let us take the first row to explain. The *tcas* program comes with 41 faulty versions. There is only one fault in each version. On average, there are 138 statements for each of the faulty versions, while only 63 to 67 of them are executable statements. Further, a test suite containing 1608 test cases is equipped, on average 2.4% of which are failed per faulty version. The other rows are similarly understood.

B. Peer Comparison

We need to compare MURE with other SBFL methods. Naish mentioned in [21], under the single fault circumstances, the *Naish2* formula is the most effective formula. Xie [32] investigated 30 risk evaluation formulas, and classified them into several groups, in which formulas are equal with each other. We select one representative technique from each group. Finally, *Naish2*, *Jaccard*, *Qe*, *Wong2*, *Wong1*, *scott*, and *M2* are used in the experiment.

To measure and compare the effectiveness of our technique and other fault localization techniques, we adopt Effectiveness metrics: Each of the techniques generates a ranked list of all the executable statements in descending order of their suspiciousness scores. Then we check all the statements along the ranked list, until a faulty statement is found.

Programmer is suggested to check statements along the ranked list, so the Expense [30] matrix can be used to measure the effectiveness of fault localization technique. The smaller the value of expense is, the more effective a fault localization technique is.

$$Expense(T) = \frac{rank\ of\ the\ faulty\ statement}{number\ of\ executable\ statements} \times 100\% \quad (4)$$

We devoted to utilize mutation analysis to improve the accuracy of SBFL rather than MBFL, so we only compare with SBFL in their fault-localization accuracies. The comparison with MBFL in time complexity to understand their efficiency has been given in Section III.D.

C. Experiment Setup

Of the 129 experiment subjects, five were not used in our experiment. Two of them, (namely, version v27 of *replace* and version v9 of *schedule2*) are excluded because all the tests pass on these versions and there is no failed test. In two versions (namely, versions v5 and v7 of *schedule*) all failed to generate mutant. Another one version (version v10 of *tcas*) is excluded for all the test cases were failed which is caused by segmentation fault. Our experimental platform (*gcc* with *gcov*) could not dump its coverage information before the runs crashed. Thus, after removing these versions, we have 124 subjects for our experiment.

For each subject, we run the tool to run the program without fault to get the right output of test cases. Then, we run the versions with faults and compare the result with the output derived from the program without fault. And we choose the statements to be mutated using the criterion described in section III.C. Then we run the mutant programs and collect the result of mutation (*fail*→*pass* or *pass*→*fail*). Then we calculate the impact value and refine the result of SBFL. At last, we compare the result derived from MURE with traditional SBFL methods, as mentioned in the next part.

D. Experiment Results

Figure 2 shows the cumulative plot of the overall effectiveness of some common fault localization technique and MURE. The x-axis indicates the code examination effort. The y-axis indicates the percentage of faults located with certain code examination efforts indicated by the x-coordinate. For each technique, a curve is used to visualize the extent of faults located within a given code examination range. For example, by examining up to 5% code in each faulty version, MURE can locate faults in 58.06% (72 of 124) of them. By examining up to 10% code in each faulty version, MURE can locate faults in 65.32% (81 of 124) of them. For each technique, the curve is drawn by connecting each point.

Figure 2 shows that the curve of MURE is always above the others, which means that, when the certain code has been examined, the MURE can always localize more faults than other techniques can. Let us take the 5% point for illustration. By examining up to 5% executable statements, Naish2 can find 51.62% of all faults, while MURE can locate faults in 58.06% of all the faulty versions. We can figure out that our technique is obviously effective than Naish2 when the examined code is not beyond 40%. Meanwhile, MURE can find over 90% of all faults when we examined 28% of all statements while Naish2 has to examine over 36% of all statements. Comparison with curves for the other peer techniques shows similar results. Limited by space, we do not explain each of them. From Figure 2, we have an impression that MURE outperforms the other techniques. However, to better understand their differences, we need other presentation of the experiment results.

To give a statistical comprehension, Table II presents the number of faults can be located by all the fault localization techniques at certain code examined expense. Let us take the first row to illustrate. It compares the localization accuracy of eight methods, including MURE and seven SBFL techniques. And the table shows the number of faults localized by different methods at given percentage of code examined. For instance, while %1 codes are examined, *Naish2* can localize 17 faults (13.7% of all faults), *Jaccard* can locate 12 faults (9.7% of all faults), *Qe* can locate 11 faults (8.9% of all faults), *Wong1* can locate 0 faults, *Wong2* can locate 7 faults (5.6% of all faults), *scott* can locate 12 faults (9.7% of all faults), *M2* can locate 15 faults (12.1% of all faults), while MURE can localize 28 faults (22.58% of all faults). By looking at the other rows, we observe similar phenomenon that MURE can locate more faults than any other technique at the same code examination percentage. We can find that MURE make full use of the advantage of MBFL and test cases change information, which is more accurate than original SBFL.

Table II also summarizes the statistics of accuracy of each technique. Take the first two rows as an example, it shows that in the best situation, all the technique can locate the fault when 0% of the code has been examined. But in the worst situation, in order to locate the fault, *Naish2* need to examine 79.20% of the statements, *Jaccard* need 85.14%, *Qe* need 86.13%, *Wong1* need 82.17%, *Wong2* need 95.05%, *scott* need 99.01%, *M2* need 84.16%, while MURE need only 62.37% to locate the fault. It shows that MURE performs

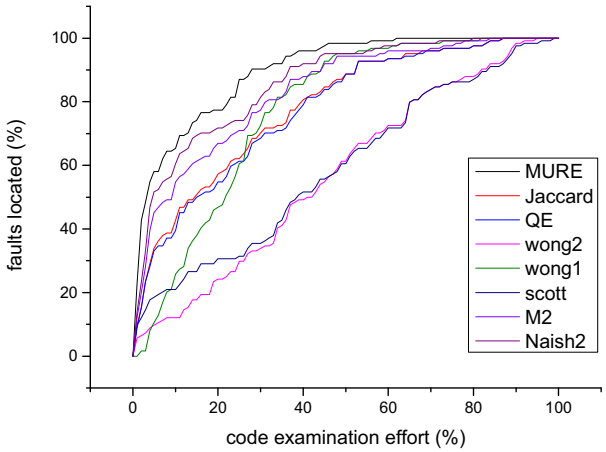

Figure 2. Comparison of different methods of fault localization

better at the bad situation than traditional situations. By looking at the other rows, we observe similar phenomenon that statistically, MURE can locate the fault by examining less code than other SBFL techniques. Let us further take the column of MURE as example. Before reaching the faulty statement, programmers have to examine from (in the best case) 0% to (in the worst 62.37%) of all statements, respectively. The median and mean code examination efforts are 3.96% and 10.25%, and the standard deviation is 13.22%. By looking at the other columns, we find that among the eight techniques, MURE always get the smallest Expense values, that means MURE always locates faults with least code examined statistically. In addition, the deviation of MURE is the lowest among these techniques, which means that MURE is relatively more stable than other techniques in various circumstances.

We further calculate how much improvement in fault-localization accuracy MURE made over a peer technique *T*. The improvement is measured as follow.

$$Improve(T) = \frac{Expense(T) - Expense(MURE)}{Expense(T)} \times 100\% \quad (5)$$

The numerator captures how much save in code examination from the peer technique *T* to MURE, by referencing the mean *Expense* of both. The denominator is the mean *Expense* of *T*. As a result, such a ratio reflects the relative improvement from *T* to MURE. The results show that MURE reduces 30.13% (i.e., 100% − (10.25% / 14.67%)) the average examination effort of *Naish*2. At the same time, *Jaccard* is improved by 52.74% in average, *Qe* is improved by 54.34%, *Wong1* is improved by 54.81%, *Wong2* is improved by 75.89%, *scott* is improved by 74.78%, and *M2* is improved by 39.17%. All these results verify that using mutant analysis to improve the accuracy of the SBFL is effective.

In summary, MURE can apparently improve the accuracy of SBFL, locating more faults than traditional SBFL methods at the same percentage of codes examined.

Table II. ACCURACY OF DIFFERENT FAULT LOCALIZATION TECHNIQUES

Code examined (%)	Numbers of faults located							
	Naish2	***Jaccard***	***Qe***	***Wong1***	***Wong2***	***scott***	***M2***	***MURE***
1	17	12	11	0	7	12	15	28
5	63	42	41	13	12	23	56	72
10	74	53	49	26	15	26	68	81
15	86	63	62	47	22	33	76	92
20	89	71	68	58	30	38	83	96
30	101	87	85	90	42	44	96	112
40	114	100	98	106	61	64	109	119
50	118	110	110	118	76	75	117	122
60	121	116	116	120	90	89	119	123
70	122	119	119	122	104	104	120	124
80	124	121	121	123	109	108	123	124
90	124	124	124	124	122	121	124	124
100	124	124	124	124	124	124	124	124
	Percentage of statements examined before localizing the fault							
min	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
max	79.20	85.14	86.13	82.17	95.05	99.01	84.16	62.37
median	4.84	14.43	14.83	21.37	40.62	38.42	9.58	3.96
mean	14.67	21.69	22.45	22.68	42.51	40.65	16.85	10.25
stdev.	18.08	21.52	21.67	15.54	26.12	28.68	19.21	13.22
	Improvements made by *MURE* over *T*							
Improve (%)	30.13	52.74	54.34	54.81	75.89	74.78	39.17	-

E. Threats to Validity

Naish2 is the optimal formula under specific condition. With such considerations, cooperating other formulas in our proposal may result in different conclusions in evaluating the effectiveness of our proposal.

The choice of parameters (like candidate number ad sampling rate) in our proposal also put threats to the validity of empirical observations.

In our experiment, when a mutant run encounters a segmentation fault, as our approach does not collect the coverage of a mutant run, this mutant run will be retained and identified as failed.

At the same time, we measured code examination effort as the amount of code to be inspected before reaching the first fault. We also realized that different experiment setups might result in different experiment results.

V. DISCUSSION

Even though our proposal has improved SBFL effectively, it still has space for improvements.

A. On Other Techniques

In this paper, we apply mutant analysis on *Naish2*, and the result shows that it significantly improves the accuracy of *Naish2*, but it does not mean that our approach can only apply to *Naish2*. A change on the formula of impact, we can get impact formulas of other SBFL. For example, the formula Hamming is $a_{ef}+a_{np}$. The final modified formula can be as follow:

$$\mathrm{a_{ef}} + \mathrm{a_{np}} + \frac{\mathrm{impact(S_i)}}{\mathrm{P}} \qquad (6)$$

Discovering the effectiveness of our proposal on other techniques and evaluating the corresponding improvement belongs to our future work.

B. Change Set Measurement

When we calculate impact, we simply compare the average size of test case sets mentioned in (1), (2). It is kind of rough to just compute the number of changed cases. For example, the original cases consist of 80 passed cases and 20 failed cases. After executing a mutant, ten test cases changes from "pass" to "fail" and three test cases change from "fail" to "pass". Suppose that balance coefficient is 3.33, the impact value will be close to zero, what if we utilize the vectorization method, and calculate the Euclidean Distance of output of different test cases, instead of scalar addition? We plan to conduct an empirical study in our future work.

VI. CONCLUSION

It is time-consuming to locate fault in a program. Existing spectrum-based fault localization (SBFL) techniques and mutation-based fault localization (MBFL) techniques estimate the location of faults in programs to narrow down the region of the fault localization. The latter is reported more accuracy in existing empirical studies, while much more exhaustive in time complexity.

In this paper, we have proposed an effective model for integrating the advantages of SBFL and MBFL. We designed MURE, which utilizes mutation analysis to refine the result of SBFL. We control the computation cost by reducing it from both number of statements as mutation candidates and number of mutates as mutation operators. We carried out a controlled experiment on seven programs from Siemens suite and compare our model with eight peer techniques, including the state-of-the-art technique *Naish2*. We have found that MURE can improve the average accuracy of all of them in comparison. In particular, the accuracy improvement over *Naish2* is over 30%.

Further work includes applying mutant technique on define-use pair or other information flows to enhance the effectiveness and safety of SBFL and extend the mutation analysis to multi-fault program.

ACKNOWLEDGMENT

This work was supported by a grant from the National Key Basic Research Program of China (project no.

2014CB340702), a grant from the National Natural Science Foundation of China (project no. 61379045), a grant from the China Scholarship Council (project no. 201604910232), an open project from the State Key Laboratory of Computer Science (project no. SYSKF1608), and grants from the General Research Fund of the Research Grants Council of Hong Kong (project nos.1120 0 015 and 11201114).

REFERENCES

[1] R. Abreu, P. Zoeteweij and A.J.C. Van Gemund. An Evaluation of Similarity Coefficients for Software Fault Localization. In *Proceedings of the 12th Pacific Rim International Symposium on Dependable Computing* (*PRDC '06*), pp. 39-46, 2006.

[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 - MUTATION (TAICPART-MUTATION '07), pp. 89-98, 2007.

[3] R. Abreu, P. Zoeteweij, R. Golsteijn, and A.J.C. van Gemund. A practical evaluation of spectrum-based fault localization. *J. Syst. Softw.*, 82(11): 1780-1792, 2009.

[4] Jr. A.T. Acree, On mutation, PhD Thesis. 1980, Georgia Institute of Technology: Atlanta, Georgia, pp. 184, 1980. T.A. Budd. Mutation Analysis of Program Test Data. Ph.D. Dissertation. Yale University, New Haven, CT, USA, pp.155, 1980.

[5] 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 '06*), pp. 82-91, 2006.

[6] T.A. Budd. Mutation Analysis of Program Test Data. Ph.D. Dissertation. Yale University, New Haven, CT, USA, pp.155, 1980.

[7] R.A. DeMillo, R.J. Lipton and F.G. Sayward. Hints on Test Data Selection: Help for the Practicing Programmer. *Computer*, 11(4): 34-41, 1978.

[8] H. Do, S. Elbaum and G. Rothermel. Supporting Controlled Experimentation with Testing Techniques: An Infrastructure and its Potential Impact. *Empirical Software Engineering*, 10(4): 405-435, 2005.

[9] P. Gong, R. Zhao, and Z. Li. Faster mutation-based fault localization with a novel mutation execution strategy. *Software Testing, Verification and Validation Workshops (ICSTW, 2015)*, pp. 1-10.

[10] R.G. Hamlet. Testing Programs with the Aid of a Compiler. *Software Engineering, IEEE Transactions on*, 3(4): 279-290, 1977.

[11] M. Hutchins, H. Foster, T. Goradia, and T. Ostrand. Experiments of the effectiveness of dataflow- and controlflow-based test adequacy criteria. In *Proceedings of the 16th international conference on Software engineering* (*ICSE '94*), pp. 191-200, 1994.

[12] Y. Jia and M. Harman. An Analysis and Survey of the Development of Mutation Testing. *Software Engineering, IEEE Transactions on*, 37(5): 649-678, 2011.

[13] J.A. Jones, M. J. Harrold, and J. Stasko. Visualization of test information to assist fault localization. In *Proceedings of the 24th International Conference on Software Engineering* (*ICSE '02*), pp. 467-477, 2002.

[14] J.A. Jones. Fault Localization Using Visualization of Test Information. In *Proceedings of the 26th International Conference on Software Engineering* (*ICSE '04*), pp. 54-56, 2004.

[15] J.A. Jones and M.J. Harrold. Empirical evaluation of the tarantula automatic fault-localization technique. In *Proceedings of the 20th IEEE/ACM international Conference on Automated software engineering* (*ASE '05*), pp. 273-282, 2005.

[16] W. Li and X. Mao. Alleviating the Impact of Coincidental Correctness on the Effectiveness of SFL by Clustering Test Cases. *Theoretical Aspects of Software Engineering Conference (TASE 2014)*, pp. 66-69.

[17] A.P. Mathur and W.E. Wong. An empirical comparison of data flow and mutation-based test adequacy criteria. *Software Testing, Verification and Reliability*, 4(1): 9-31, 1994.

[18] B. Marick, The Weak Mutation Hypothesis, In Proc. of ISSTA'91, Page 190-199, Oct., 1991.

[19] B. Marick, Faults of Omission, Software Testing and Quality Engineering Magazine, 2(1), 2000.

[20] S. Moon, Y. Kim, and M. Kim. Ask the Mutants: Mutating faulty programs for fault localization. 2014 IEEE International Conference on Software Testing,Verification, and Validation.

[21] L. Naish, H. J. Lee, and K. Ramamohanarao. A model for spectra-based software diagnosis. *ACM Transaction on Software Engineering Methodology*, 20(3): 11, 2011.

[22] A.J. Offutt and R.H. Untch. Mutation 2000: Uniting The Orthogonal. In *Proceedings of the 1st Workshop on Mutation Analysis* (*MUTATION' 00*), pp. 34–44, 2001.

[23] M. Papadakis and Y. Le Traon, "Using Mutants to Locate "Unknown" Faults," 2012 IEEE Fifth International Conference on Software Testing, Verification and Validation, Montreal, QC, 2012, pp. 691-700.

[24] M. Renieris and S.P. Reiss. Fault localization with nearest neighbor queries. In *Proceedings of the International Conference on Automated Software Engineering*, pp. 30-39, 2003.

[25] H.A. Richard, R.A. Demillo, B. Hathaway, W. Hsu, W. Hsu, E.W. Krauser, R.J. Martin, A.P. Mathur, and E.H. Spafford. Design Of Mutant Operators For The C Programming Language, Technical Report SERC-TR-41-P, Software Engineering Research Center, Purdue University, 1989.

[26] D.J. Richardson and M.C. Thompson. An analysis of test data selection criteria using the RELAY model of fault detection. *Software Engineering, IEEE Transactions on*, 19(6): 533-553, 1993.

[27] R. Santelices, J.A. Jones, Y. Yu, and M.J. Harrold. Lightweight fault-localization using multiple coverage types. In *Proceedings of the 31st International Conference on Software Engineering* (*ICSE '09*), pp. 56-66, 2009.

[28] X. Wang, S.C. Cheung, W.K. Chan, and Z. Zhang. Taming coincidental correctness: Coverage refinement with context patterns to improve fault localization. In *Proceedings of the 31st International Conference on Software Engineering* (*ICSE '09*), pp. 45-55, 2009.

[29] W.E. Wong and Y. Qi. Effective program debugging based on execution slices and inter-block data dependency. *J. Syst. Softw.*, 79(7): 891-903, 2006.

[30] W.E. Wong, Y. Qi, L. Zhao, and K-Y Cai. Effective Fault Localization using Code Coverage. In *Proceedings of the 31st Annual International Computer Software and Applications Conference* (*COMPSAC '07*), Vol. 1, pp. 449-456, 2007.

[31] E. Wong, V. Debroy and B. Choi. A family of code coverage-based heuristics for effective fault localization. *J. Syst. Softw.*, 83(2): 188-208, 2010.

[32] X. Xie, T.Y. Chen, F. Kuo, and B. Xu. A theoretical analysis of the risk evaluation formulas for spectrum-based fault localization. ACM *Transactions on Software of Engineering and Methodology*(*TOSEM 2013*),pp.1-40.

[33] X.Y. Zhang, Z. Zheng, and K.Y. Cai. (2017). "Exploring the usefulness of unlabelled test cases in software fault localization". Journal of Systems and Software (JSS 2017), pages 1–13.

[34] L. Zhang, L. Yan, Z. Zhang, J. Zhang, W. K. Chan, and Z. Zheng. A theoretical analysis on cloning the failed test cases to improve spectrum-based fault localization. *Journal of Systems and Software (JSS)*, Volume 129, July 2017, Pages 35-57.

[35] Software testing help. "Software Testing Terms- Complete Glossary", available on: http://www.softwaretestinghelp.com/software-testing-terms-complete-glossary/.