Isolating Suspiciousness from Spectrum-Based Fault Localization Techniques

Xiaoyuan Xie[1, 2, *], Tsong Yueh Chen[1], Baowen Xu[3]

[1]Centre for Software Analysis and Testing
Swinburne University of Technology
Hawthorn, Victoria 3122 Australia
{xxie, tychen}@groupwise.swin.edu.au

[2]School of Computer Science and Engineering
Southeast University
Nanjing 210096, China

[3]State Key Laboratory for Novel Software Technology& Department of Computer Science and Technology
Nanjing University
Nanjing 210093 China
bwxu@nju.edu.cn

***Abstract*—Spectrum-based fault localization (SBFL) is one of the most promising fault localization approaches, which normally uses the failed and passed program spectrum to evaluate the risks for all program entities. However, it does not explicitly distinguish the different degree in definiteness between the information associated with the failed spectrum and the passed spectrum, which may result in an unreliable location of faults. Thus in this paper, we propose a refinement method to improve the accuracy of the predication by SBFL, through eliminating the indefinite information. Our method categorizes all statements into two groups according to their different suspiciousness, and then uses different evaluation schemes for these two groups. In this way, we can reduce the use of the unreliable information in the ranking list, and finally provide a more precise result. Experimental study shows that for some SBFL techniques, our method can significantly improve their performance in some situations, and in other cases, it can still remain the techniques' original performance**

Keywords - program spectrum; fault localization; debugging; risk evaluation

I. INTRODUCTION

It is commonly recognized that testing and debugging are important but resource consuming activities in software engineering. Trying to locate the faults is one of the most essential but tedious tasks, due to a great amount of manual involvement. Therefore, many researchers aim at provoking automatic and effective fault localization technique, in order to decrease its cost under the limited resources, as well as to increase the software reliability.

One promising approach for automatic fault localization is Spectrum-based Fault Localization (referred to as SBFL). Generally speaking, this approach utilizes the dynamic testing results and various program spectra information acquired from the testing to evaluate the risk of containing a fault for each program entity with different statistical formulas, and finally to give a risk ranking list [1]. Some typical techniques include Pinpoint [3], Tarantula [4], Ochiai [5] and etc. [6-12].

SBFL approach has gained much popularity due to its nature of simplicity and practicality. However there are still some problems in this technique. One of them is related to the definiteness of the information used. By the limitation of software testing, SBFL has such characteristic that: information associated with program spectra of failed test cases and passed test cases has different degree of definiteness. Failed program spectrum has absolutely definite information that it must contain at least one faulty entity. On the other hand, a passed spectrum is not guaranteed to be absolutely free of any faulty entity. Therefore, in SBFL, the passed spectra must be utilized together with the failed ones, to provide a more comprehensive picture about the risks of the statements. For a program entity with no failed information, we cannot draw any conclusion about its correctness.

However, even though some statistical formulas in SBFL may implicitly conceive that risk evaluation should avoid using the passed information alone; none of them has explicitly pointed out this idea, and in fact some SBFL techniques do overlook this idea. Take *Hamming Distance*, as an example, which uses two indexes for measuring the risk for each program entity, that is, the number of failed test cases which have executed this entity and the number of passed test cases which have never touched the entity [13]. The sum of these two indexes is regarded as the risk. However, for certain program entity which has never been executed by any failed test cases, the first index is 0. This formula becomes solely dependent on the second index, which is equivalent to using information provided by the passed spectra only. In such a situation, the calculated risk becomes less reliable.

Thus, in this paper, we propose a refinement method to improve the accuracy of the predication by SBFL, through eliminating the indefinite information for some statistical risk evaluation formulas. We use the statement-hit spectrum as an illustration, which is widely used by SBFL community. Given a SBFL technique, rather than evaluating each statement's risk of being faulty with a relevant statistical formula, our proposal first categorizes all statements into suspicious group and unsuspicious group. The suspicious group contains statements which have shown up at least once in a failed spectrum, that is, they have been demonstrated to have a chance of being faulty; while in the unsuspicious group, no statement has shown up in any failed test run. Intuitively speaking, these statements cannot be the faulty ones that have been "*activated*" by the current test suite. After the categorization, we then continue the normal risk evaluation using the original statistical formula on the suspicious statements; meanwhile assign all unsuspicious statements with the lowest risk value. In this way, we can improve the reliability of the ranking result, and statements in

* Corresponding author

1550-6002/10 $26.00 © 2010 IEEE
DOI 10.1109/QSIC.2010.45

IEEE computer society

suspicious group are given a higher priority in debugging, as they are deserved to have.

Our proposal seems to be simple and obvious, but it is important because we explicitly summarize a simple but essential idea into a basic rule, which should be considered in the further SBFL technique design. Additionally, for SBFL techniques with the above problem, this method can serve as a remedy and only incurs minimal overhead. It is intuitively obvious that our method can enhance the performance of such SBFL techniques, but the significant improvement is somehow a pleasant surprise as would be demonstrated later.

The rest of this paper is organized as follows: Section II describes the background of the spectrum-based fault localization. Section III discusses the problem due to the use of less definite information with a simple example, and also describes our method in details. In Section IV, we introduce the experimental setup. Section V presents the empirical results with the extensively used *Siemens Suite*, and provides some analysis about the effectiveness of our proposal. In Section VI, we present the related works and compare our method with them. Finally we present the conclusion and the potential problems for further study in Section VII.

II. BACKGROUND: SPECTRUM-BASED FAULT LOCALIZATION

Spectrum-based fault localization, referred to as SBFL, is a dynamic approach. Two essential types of information are collected for SBFL, namely testing results and program spectrum. Testing results are recorded as either *pass* or *fail*. While a program spectrum records the run-time profiles about various program entities for a specific test suite [7]. These entities could be individual statements, branches, etc; while the run-time information could be the binary coverage status, the time that the entity has been executed, etc. Practically there are many kinds of combinations [1]. In this paper, we will use the commonly adopted spectrum, statement-hit spectrum, which collects the binary execution flag for each statement, as an illustration.

Utilizing these two types of information, SBFL produces a vector which consists of four indexes for each statement, denoted as A=<a_{ef}, a_{ep}, a_{nf}, a_{np}>, where a_{ef} and a_{ep} denote the number of test cases that execute the corresponding statements with a failed and passed result respectively. While a_{nf} and a_{np} represent the number of test cases that do not execute the corresponding statement, but return a failed and passed result respectively. An example is shown in Figure 1.

$$P:(s_1 \quad s_2 \quad s_3 \quad s_4)$$

$$TS:\begin{pmatrix} t_1 \\ t_2 \\ t_3 \\ t_4 \\ t_5 \\ t_6 \end{pmatrix} M_S:\begin{pmatrix} 1 & 0 & 0 & 1 \\ 1 & 0 & 1 & 1 \\ 1 & 0 & 1 & 0 \\ 1 & 1 & 1 & 1 \\ 1 & 0 & 0 & 1 \\ 1 & 0 & 1 & 1 \end{pmatrix} RE:\begin{pmatrix} e_1: pass \\ e_2: pass \\ e_3: pass \\ e_4: pass \\ e_5: fail \\ e_6: fail \end{pmatrix}$$

$$A:\begin{pmatrix} a_{ef} \\ a_{ep} \\ a_{nf} \\ a_{np} \end{pmatrix} M_A\begin{pmatrix} 2 & 0 & 1 & 2 \\ 4 & 1 & 3 & 3 \\ 0 & 2 & 1 & 0 \\ 0 & 3 & 1 & 1 \end{pmatrix}$$

Figure 1. Example of a program spectrum.

It can be seen from Figure 1 that there are 4 statements <s_1, s_2, s_3, s_4> in the current program P. And a test suite TS, consisting of 6 test cases <t_1, t_2, t_3, t_4, t_5, t_6> is executed, two of which (t_5 and t_6), give rise to failed runs and the other four give rise to passed runs. Vector RE = <e_1, e_2, e_3, e_4, e_5, e_6> lists the testing results of the corresponding test cases, where e_i is the testing result for t_i. Matrix M_S gives the statement-hit spectrum. The binary number in each cell c_{ij}^{Ms} represents the coverage information of statement s_j, by the test case t_i, with 1 if s_j is executed, and 0 otherwise. Matrix M_A is defined such that each of its column sub-vectors M_A^j represents the vector A for statement s_j. For instance, in Figure 1, a_{np} = 0 for s_1 means that no test case in the current test suite can both pass and skip s_1. And a_{ef} =2 for s_4 represents that s_4 is executed by two test cases which can detect failure. Of course the sum of the four indexes for each statement should be equal to the size of the test suite.

Apparently the ultimate goal for SBFL is to highlight the parts of the program whose activities strongly correlate with failures. Thus a risk evaluation formula R is provided, which is the most essential part in SBFL, to project the vector M_A^j into its statement risk value r_j for s_j. Normally for a risk evaluation formula, statements with higher risk values are considered more likely to contain faults. After getting the risk value vector V_R = <r_1, r_2, r_3, r_4> evaluated by formula R for all the four statements, testers debug by inspecting the statements according to the descending order of V_R.

Suppose program P contains n statements <s_1, s_2, …, s_n>, then the risk evaluation formula R: $D_A \rightarrow D_R$, where D_A contains the vector A=<a_{np}, a_{nf}, a_{ep}, a_{ef}> for all statements, that is, D_A={ $M_A^j \mid M_A^j = < a^j_{ef}, a^j_{ep}, a^j_{nf}, a^j_{np} >$, $1 \le j \le n$ }, and D_R contains all possible risk r, whose value could be any real number according to different definitions.

In SBFL, different formulas have different definitions of R, from the long established Hamming metric, which was originally introduced for error detecting and correcting codes in 1950 [13], to recently adopted or proposed ones by SBFL community, including Jaccard[3], Ochiai[5], Ample[8], Tarantula[4], Wong's metrics [12], etc. Generally they are developed from different perspectives or serve for different purposes. For example, the Tarantula system [4] has been developed for statement-hit spectra, instead of the previously used block-hit spectra. Together with a visualized report, it gives a more detailed and practical diagnosis. While Wong et. al [12] provide some more reasonable metrics by distinguishing the effects of different passed test cases in risk evaluation. These metrics are based on the heuristic that the impact of the first passed test case in evaluating the risk of certain statement is more significant than or equal to that of the second passed test case that executes it. Anyhow no matter from what perspective are the formulas derived, they should all follow the same intuition that the faulty statements must be related to failed runs more closely than the correct statements. Since the evaluation of a specific formula is not the focus of this paper, we will not discuss this in details and just simply adopt those formulas, and see how much they can be improved by our refinement method.

III. METHODOLOGY

A. Assumption

Generally speaking, the performance of SBFL is solely dependent on the test suite. Therefore in this paper, all the "*faulty statements*" that we refer to are meant for the ones that have been "*activated*" by the current test suite. For those "*sleeping*" faults which have not yet been revealed by any failed test cases are not considered in our discussion. This assumption is meaningful because only failed runs can give us definite indication about the faults in the program. And SBFL techniques just utilize this information to evaluate the risk for each statement of being faulty with various statistical formulas. Hence, if we want to discover more faults, the only way is to improve the current test suite, getting at least one test case revealing the additional faults, and hence acquire some definite information about them. Actually this is also assumed implicitly by all existing SBFL techniques, which require a test suite with at least one failed test case.

B. Problem

SBFL uses two types of program spectra, the failed and the passed spectrum, to estimate the risk of each statement, with different risk evaluation formulas. However, the information associated with these two types of spectra has different degrees of definiteness. Failed program spectrum has absolutely definite information that it must contain at least one faulty statement. On the other hand, a passed spectrum is not guaranteed to be absolutely free of any faulty statement. Therefore with the use of passed spectra information alone we cannot draw any reliable conclusion about the correctness of certain statement.

Normally SBFL techniques intend to utilize both kinds of information to make a decision. However sometimes it may happen to use only the passed spectrum to evaluate the risk for certain statement. This may yield an unreliable risk value, and consequently results in a less precise localization result.

Here we still use the example in Figure 1 to illustrate this situation. Suppose in the program P, s_4 is the sole faulty statement. Let us take one of the Wong's formulas as an example: $R(A): r = a_{ef} - a_{ep}$ [12]. According to each column vector M_A^j in M_A, we get the vector V_R = <-2, -1, -2, -1>, which contains the risk value for each statement correspondingly. It can be seen that s_2 has got the same highest risk as the real faulty statement s_4. The high risk of s_2 is due to the presence of indefinite information associated with a_{ep}, as $a_{ep} = 1$, but not the definite information associated with a_{ef}, as $a_{ef} = 0$. Therefore in this paper, we propose a refinement method by excluding such scenarios.

C. Solution

Suppose the statement set of program P is $\{s_1, s_2, \ldots, s_n\}$. Since each failed spectrum can definitely indicate the presence of at least one "*activated*" faulty statement, based on the assumption in Section III.A, we can easily conclude that all the "*activated*" faulty statements must reside and only reside in the union of all failed spectra. Therefore we can categorize all statements into two groups: the suspicious group G_s and the unsuspicious group G_u. We define $G_s = \bigcup_{failed} tr_i$, where $tr_i = \{s_j \mid s_j$ such that it is executed by test case $t_i\}$, that is, the collection of statements with value 1 in the statement-hit spectrum for test case t_i. And consequently we define $G_u = P - G_s$.

It can be seen from the definition that G_s contains all statements which have shown up at least once in a failed spectrum, that is statements with $a_{ef} \neq 0$; while in G_u, no statement has shown up in any failed test run, that is statements with $a_{ef} = 0$. It is obvious that with respect to the current "*activated*" faults, the statements in G_s have been demonstrated to have a chance of being faulty. While the statements in G_u are clean, that is, they cannot be the "*activated*" faulty statements. For the example in Section III.B, we have $G_s = \{s_1, s_3, s_4\}$ and $G_u = \{s_2\}$.

After the above categorization, we update the evaluation formula with $R^{new}: D_A \rightarrow D_R$ in the following way:

$$R^{new}(A_i) = \begin{cases} R(A_i) + 1 & \forall s_i \in G_s \\ Min_risk & \forall s_i \in G_u \end{cases}$$

where $A_i = < a^i_{ef}, a^i_{ep}, a^i_{nf}, a^i_{np} >$ denotes the vector A for statement s_i, and $Min_risk = min\{R(A_i) | \forall s_i \in G_s\}$, that is the minimal risk value among all statements $s_i \in G_s$. For the example in Section III.B, after using our refinement method, the new risk vector becomes V_R = <-1, -2, -1, 0>. This time, the SBFL technique makes a more accurate estimation.

It can be seen that formula R^{new} evaluates the risks of statements in different groups with different schemes, which can separate them in the final ranking list. For statements in G_s, the new risk values are their original values acquired from the original formula R, added by a constant. The actual value of this constant is not important, since only the relative order among all statements is important; hence the aim of this addition is just to assure that all statements in G_s have higher risk values than statements in G_u. In our definition of R^{new}, we use 1 as an example.

On the other hand, for statements in G_u, the new risk values are simply assigned with the minimum among all original risk values evaluated by R. They are not distinguished in our method because the only available information about these statements from the current test suite is the number of passed test cases that have executed these statements. There are 2 possible situations:

1. $a_{ep} = 0$, that is this statement has never been executed by any test case in the current test suite. In such a case, we actually have no information about this statement.

2. $a_{ep} \neq 0$, which means that this statement has been executed, but only by some passed test cases. According to the limitation of software testing, the passed runs themselves cannot provide any absolutely definite information about the correctness of the statements.

However a_{ep} may be affected by the current test suite, hence with this index only, we cannot reliably distinguish these statements. Actually the scheme that assigns the risk of statements in G_u with the same lowest value indicates that these statements can be ignored in debugging, because we are

absolutely sure that they can never be any of the current "*activated*" faulty statements. Even they may be the "*sleeping*" faults; we still have no way to be sure what they actually are. Hence the most reasonable way is to ignore them in debugging, until they are executed by some new failed test cases. At that time, they become suspicious statements, with respect to the newly "*activated*" faults revealed by the new failed test cases.

In a word, with R^{new} all statements in G_s will be ranked at the top of the final list, in their originally relative order ranked by R. While all statements in G_u will be ranked at the bottom of the list. In such a way, our method can provide a more precise localization result by excluding the indefinite information and make programmers to focus on the statements in G_s. Of course, we can improve R^{new} by distinguishing statements in G_u, apparently, with some additional information that can provide definite or heuristic hints. And this will be investigated in our future study.

It can be seen that our method helps to distinguish the difference of information associated with failed test cases and passed test cases, in terms of their definiteness. Actually among all the existing statistical risk evaluation formulas in SBFL, which are derived from different intuitions and serve for various purposes, some may implicitly conceive that risk evaluation should avoid using the passed information alone (for example the Tarantula [4], but not Wong's formula [12]). However none of them has explicitly pointed this out, which makes it easily be overlooked. Hence our proposal explicitly summarizes this simple but essential idea into a basic rule, which should be considered in the design of new SBFL techniques. More importantly, for those existing formulas which do not conceive such idea, our method can serve as an effective refinement remedy with marginal overhead. Of course, the effectiveness could be different with different fault types and test suites. Thus, it is interesting for us to investigate how effective our method can be.

IV. EXPERIMENTAL SET UP

A. Testing objects

In our case study, we use the *Siemens suite* as our benchmark, which is acquired from the Software Information Repository [15]. The *Siemens suite* contains seven small programs, several faulty versions of these programs and also a series of test suites for various testing criteria. Table I lists the programs, the number of faulty versions of each program, lines of code, number of all test cases, as well as a brief description about the functionality of the corresponding program. For more detailed information, please refer to [16].

We choose the *Siemens suite* simply because it is a widely used benchmark for fault localization community [5, 12, 17, 18]. In our experiments, we utilize the "*universe*" suite, which contains all the test cases in the Table I. However we cannot adopt all mutants of these programs (132 versions in total). Among these 132 mutants, version 10 of *print_tokens2*, version 32 of *replace* and version 9 of *schedule2* have no failures detected by any test case, hence they are not considered by our experiments. Besides, in our study, we focus on the single-fault mutants. Hence, version 1 of *print_tokens*, version 21 of *replace*, version 2 and version 7 of *schedule*, versions 10, 11, 15, 31, 32, 33 and 40 of *tcas* are discarded in the experiments. Furthermore, in our experiments, we aim to investigate the executable statements, thus we ignore the modification in non-executable statements, such as changing in the header files, mutants in variable declaration statements, or modifications in a macro statements started with #define. For this purpose we further exclude the following mutants: version 12 of *replace*, versions 2, 4 and 6 of *print_token*, versions 13, 14, 36, 38 of *tcas*, and versions 6, 10, 19, 21 of *tot_info*. In summary, we have excluded 26 mutants in total, and used 106 versions for experiments. Besides *gcov* is applied for the statement-hit spectrum collection.

TABLE I. BRIEF INTRODUCTION OF *SIEMENS SUITE*

Program	Versions	LOC	Number of Test Cases	Description
print_tokens	7	563	4130	Lexical analyser
print_tokens2	10	508	4115	Lexical analyser
replace	32	563	5542	Pattern recognition
schedule	9	410	2650	Priority scheduler
schedule2	10	307	2710	Priority scheduler
tcas	41	173	1608	Altitude separation
tot_info	23	406	1052	Information measure

B. Formulas under investigation

As described in Section III.C, different risk evaluation formulas are designed from different perspectives, and used for different purpose. Some of them may make use of the indefinite information. Our experimental study is focused on such formulas. Table II lists all the formulas we have investigated in the experiments, which can be divided into two groups [14].

TABLE II. INVESTIGATED FORMULAS

Name	Formula
Wong[2]	$a_{ef} - a_{ep}$
Wong[3]	$a_{ef} - h$, where $h = \begin{cases} a_{ep} & (a_{ep} \le 2) \\ 2 + 0.1(a_{ep} - 2) & (2 < a_{ep} \le 10) \\ 2.8 + 0.001(a_{ep} - 10) & (a_{ep} > 10) \end{cases}$
O	$-1 (a_{nf} > 0)$ $a_{np} (a_{nf} \le 0)$
O[p]	$a_{ef} - \frac{a_{ep}}{a_{ep} + a_{np} + 1}$
CBI	$\frac{a_{ef}}{a_{ef} + a_{ep}} - \frac{a_{ef} + a_{nf}}{a_{ef} + a_{nf} + a_{np} + a_{ep}}$
Scott	$\frac{4a_{ef}a_{np} - 4a_{nf}a_{ep} - (a_{nf} - a_{ep})^2}{(2a_{ef} + a_{nf} + a_{ep})(2a_{np} + a_{nf} + a_{ep})}$
GMean	$\frac{a_{ef}a_{np} - a_{nf}a_{ep}}{\sqrt{(a_{ef} + a_{ep})(a_{np} + a_{nf})(a_{ef} + a_{nf})(a_{ep} + a_{np})}}$
Rogot	$\frac{1}{4}(\frac{a_{ef}}{a_{ef} + a_{ep}} + \frac{a_{ef}}{a_{ef} + a_{nf}} + \frac{a_{np}}{a_{np} + a_{ep}} + \frac{a_{np}}{a_{np} + a_{nf}})$
M	$\frac{a_{ef} + a_{np}}{a_{nf} + a_{ep}}$

The first group contains formulas which have been used in other SBFL techniques. These formulas include: *Wong*^{2}, *Wong*^{3} which were originally introduced by [12], *Optimal* and its variant *Optimal*^{p} (referred to as O and O^p respectively in the table), which are provided in [14] and have been proved to be the best in certain cases among many formulas, and Cooperative Bug Isolation system (referred to as *CBI* in the

table) [10]. However, *CBI* was not originally designed for statement-hit spectrum. It is for the method called instrumentation of predicates, which instruments predicates in selected parts of the code. Hence it uses the sampling of the coverage information collected by the instrumented predicates, rather than using the whole execution trace. But its formula can be equally applied to the statement-hit spectrum.

We also select several metrics from other research domains to form the second group. Even they are not originally designed for fault localization; it is still worth to investigate them. Because most of the formulas used in fault localization are actually brought from other research areas, such as mathematics, data mining, bioinformatics, etc. Among which *Scott* [21], Geometric Mean (referred to as *GMean* in the table) [22] and *Rogot* [23] are introduced from the area of biometrics. And one of the unnamed metrics that is referred to as *M* in [14] is previously used for classification and clustering [24].

C. *Evaluation metric*

In our experimental study, we use the established measurement, relative ranking of faulty statement, to investigate the effectiveness of the application of our refinement method to some SBFL techniques. The relative ranking, referred to as p_r in this paper, is the percentage of the program that needs to be examined before a bug is found (that is the absolute ranking of the faulty statement divided by the total number of statements). Similar effectiveness measurements have been adopted by most of the previous studies in SBFL [2, 5, 6, 14, 17, 18, 25].

In our experiments, for the statements with the same assigned risks value, we rank them according to their original order in the source code. This is reasonable and practical because in the real debugging process, for a set of statements of the same risk, we have no way to specify certain checking order without any other information. Thus the most natural way is to check them one by one in their original order.

In our empirical data, we also exclude all the non-executable lines in source code, such as the comments, blank lines, the braces, declarations, macro definition (#define), etc. This is a reasonable process because they will not be considered when developers debug their program.

V. RESULTS AND ANALYSIS

A. *Effectiveness*

In our experiments, we apply the refinement method to all the formulas listed in Table II, investigating its effectiveness in improving the performance of these SBFL techniques. For each program, we use a box-plot diagram to visually present the performance improvement after adopting our refinement method in all the formulas, that is, the percentage of the relative decrease in p_r, in Figure 2 to Figure 8. From the bottom to the top, each column of these diagrams presents the minimum, the 1st quartile, the medium, the 3rd quartile, and the maximum of the percentages among all the mutants of the respective program, with certain risk evaluation formula. Columns without any data demonstrated indicate that the respective formula's performance remains unchanged after adopting our method.

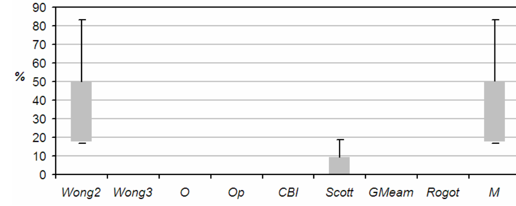

Figure 2. Performance comparison in *print_token*

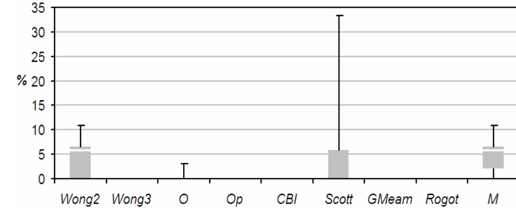

Figure 3. Performance comparison in *print_tokens2*

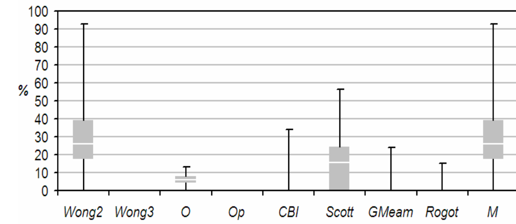

Figure 4. Performance comparison in *replace*

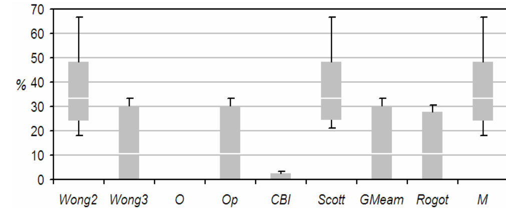

Figure 5. Performance comparison in *schedule*

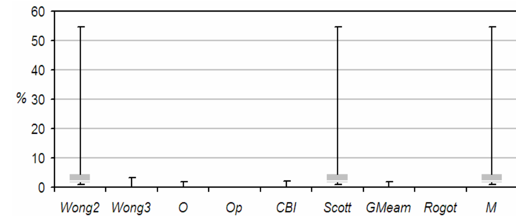

Figure 6. Performance comparison in *schedule2*

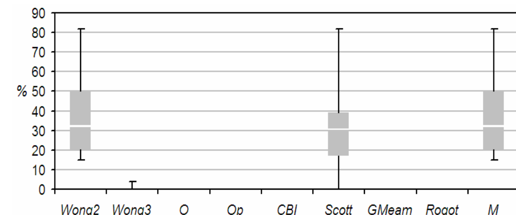

Figure 7. Performance comparison in *tcas*

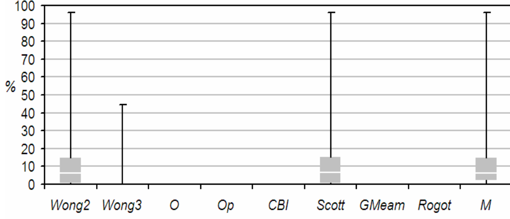

Figure 8. Performance comparison in *tot_info*

It can be seen from these figures that our method is helpful in improving the performance of SBFL in many situations: among all the programs under testing, the number of formula whose performance has been improved varies from three to eight out of nine under investigation. Especially in some mutants of these programs, some risk evaluation formulas can achieve quite significant improvement. For example, in the version 3 of *tot_info*, when using $Wong^2$ as the risk evaluation formula, our refinement method can achieve as high as 96.0% improvement. Besides for the situations where the performance of certain formula cannot be improved, our method can still ensure to remain its original p_r. Table III and Table IV summarizes the result for individual program and formula respectively.

TABLE III. SUMMARIZED IMPROVEMENT FOR INDIVIDUAL PROGRAM

Program	R_Num	Max_Imp	Min_ Imp
print_tokens	3	24.1%	10.4%
print_tokens2	4	6.3%	2.5%
replace	7	25.0%	3.6%
schedule	8	28.3%	1.0%
schedule2	7	6.7%	0.2%
tcas	4	29.4%	0.6%
tot_info	4	24.8%	3.2%

TABLE IV. SUMMARIZED IMPROVEMENT FOR INDIVIDUAL FORMULA

Formula	P_Num	Max_Imp	Min_ Imp
$Wong^2$	7	29.4%	6.3%
$Wong^3$	4	24.9%	0.2%
O	3	6.5%	0.4%
O^p	1	24.8%	24.8%
CBI	3	4.5%	0.3%
Scott	7	28.3%	5.2%
GMean	3	24.2%	0.3%
Rogot	2	23.0%	3.6%
M	7	29.4%	4.3%

In Table III, for each individual program, this table presents the number of formulas that have been improved in column "R_Num"; as well as the maximum and minimum of the average performance improvement over all the mutants among all the investigated formulas, in column "Max_Imp" and "Min_Imp" respectively. It can be seen that for program *print_tokens*, *print_tokens2*, *tcas* and *tot_info*, performance have been improved in about half of the formulas, and for program *replace*, *schedule* and *schedule2*, almost all the formulas have been improved. Among all the improvement, the highest one is in program *tcas*, using both formula $Wong^2$ and *M*. The magnitude is as high as 29.4%. Apart from *tcas*, for other programs like *print_tokens*, *replace*, *schedule* and *tot_info*, the maximal decrease in p_r is also no less than 24%. However for *print_tokens2* and *schedule2*, the improvement is not as significant as the other programs.

In Table IV, for each individual formula, column "P_Num" represents the number of programs that have been improved for certain formula. It can be discovered that different formulas have different sensitivities to our refinement method, but they have been improved in at least one program. $Wong^2$, *Scott* and *M* are of the highest sensitivity to our method, since their performances have been improved in all the 7 programs. Meanwhile, their maximal decrease in p_r is not less than 28%. Besides, in formula $Wong^3$, O^p, *GMean* and *Rogot*, even their performance have only been improved in less than 4 of the programs, their maximal magnitudes are still very substantial (no less than 23.0%). Especially for formula O^p, we surprisingly discover that in program *schedule*, it can obtain a relatively significant decrease of p_r by 24.8%. More importantly, O^p has been proved as an optimal formula in most situations [14]. On the other hand, formula *O* and *CBI* can be regarded as less sensitive to our methods, due to their low P-Num and less significant improvement than the other formulas.

B. *Threats to validity*

There are a number of threats to the validity of this experimental study. Specifically, the primary one is the representative of our results acquired from the *Siemens suite*. Although these 7 programs are adopted widely in SBFL community as a benchmark, we still need to use some more programs with larger scales, to investigate the effectiveness of our method. Besides, the results in our experiments are obtained from mutation analysis of single fault. To investigate the effectiveness of our method in a more generalized situation, we need to adopt mutants with multiple faults. Another threat is about the effectiveness measurement. We rank all statements with the same risk in their original order in the source code, but there also exist some other schemes in ranking these statements, such as average, best-worst, etc [25]. We can investigate the effectiveness of our method with these schemes. These further experiments will be completed in our future study.

VI. RELATED WORK

Apart from SBFL, current research in fault localization and debugging has developed many other techniques using various intuitions. Some of them consider only statements executed in the failed runs. Two typical ones are methods that isolate faulty statements via predicate switching and value replacement [26, 27].

In the method using predicate switching, a predicate's outcome is forcibly switched at runtime. Consequently the control flow is altered and the program state is modified. This method tries to isolate the faults by examining the predicates whose switching can bring the program execution into a successful completion, using the original failed test input. Similarly the method using value replacement aims to search for statements whose replacement will change a failed input into a passed one. This is done by replacing values used at a statement during the runtime execution of the failed test case.

It is obvious that these methods are different from our refinement method. Firstly, our method needs to be integrated with a SBFL technique, which uses information obtained from testing and various statistical formulas to evaluate and rank the risks for different program entities. The above two methods are not SBFL techniques and are derived from different perspective and intuition, using different mechanism to isolate the faults. Secondly, our method focuses on using all the failed test information to refine some risk evaluation formulas in SBFL, so as to decrease the risk value of statements which are not executed by any failed test cases. For statements which are executed by some failed test cases, we still use all the failed and passed testing information in the current test suite to evaluate their risks. However, the above two methods only use the information obtained from one single failed test case, and try to turn this "failed" test input into a "passed" one, through the predicate switching or value replacement.

Our method is also different from the set-based debugging techniques, which try to isolate the faulty statements based on various heuristic models. Two representatives are the Nearest Neighbor technique [17] and the Execution Slicing based technique [
].

The Nearest Neighbor technique has two heuristic models, namely Set-union and Set-intersection. Both of them use a single failed spectrum (tr^f) and all the passed spectra (tr^p). Set-union model aims at removing the union of all passed spectra from the failed spectrum, that is, $tr^f - \bigcup_i tr_i^p$. It focuses on statements that only belong to the failed run. Set-intersection model uses set $\bigcap_i tr_i^p - tr^f$, of which non-membership is discriminant for the fault.

The Execution Slicing based technique locates the faulty statements by using three heuristic models on execution slicing. The first model uses one failed execution slicing (ex^f), and one passed execution slicing (ex^p). The target statements are the execution dicing, ex^f-ex^p. These statements are given highest priority in debugging, while statements in set $ex^f \cap ex^p$ have the second highest priority and statements in ex^p-ex^f are assigned with lowest priority. The second model uses one failed execution slicing ex^f, and two passed execution slicing ex_1^p and ex_2^p, in which, set $ex^f - (ex_1^p \cup ex_2^p)$ is assigned with the highest priority, set $(ex^f \cap ex_1^p) - ex_2^p$ and $(ex^f \cap ex_2^p) - ex_1^p$ are equally assigned with the second highest priority, set $ex^f \cap ex_1^p \cap ex_2^p$ are assigned with the third highest priority, while the remaining statements obtain the lowest priority. And the third model, as well as the most general case, uses multiple failed and multiple passed execution slicing. Suppose the whole test suite is *TS*, for any of its subset *TS'* which contains both failed and passed test cases, the construction scheme is $\bigcap_i ex_i^f - \bigcup_j ex_j^p$, where ex_i^f and ex_j^p are the failed and passed execution slicing of test cases that belong to *TS'* respectively.

However it can be seen that, none of the above models has explicitly distinguished the different degree of reliability between the information associated with the failed and the passed test cases. Hence they can provide only some heuristic results. Meanwhile, apart from the third model in Execution Slicing based technique, all the other models are based on only one failed test case with definite information and multiple passed test cases with indefinite information, which makes the results even more unreliable. Differently, our method explicitly makes use of the distinction between definite information of failed runs and indefinite information of passed runs. Based on this fact, we use all the failed testing information to categorize statements.

Furthermore these techniques only provide a set of statement, which serves as the initial suspicious set for debugging. However they do not clearly distinguish the suspiciousness either within the set or outside the set, that is, they cannot provide a more subtle risk ranking list. Even though the first two models of the Execution Slicing based technique can assign different sets with different priorities, for the third model in general case, since the combination of execution slicing becomes much more complex, it is not a trivial task of assigning priorities to all these combinations.

While our method, which serves as an enhancement to SBFL, can fully utilize the information from the current test suite and evaluate the risks for suspicious statements by adopting the same method of SBFL. Thus it not only helps to exclude the unsuspicious statements, but also can supply a subtle ranking list for all suspicious statements, which can facilitate the debugging much better.

VII. CONCLUSION

In this study, we proposed a refinement method for SBFL techniques. In our proposal, we take into consideration of the distinction between the definiteness of the information carried by the failed spectra and the passed spectra, and hence improve the accuracy of the predication by some statistical risk evaluation formulas, through the elimination of indefinite information. Our approach works under the assumption that SBFL process should focus on those "*activated*" faulty statements with respect to the current test suite; other potential "*sleeping*" faults should be ignored. This is generally assumed by the SBFL community, since all of its existing techniques require a test suite with at least one failed run to reveal the fault.

Under the above assumption, we categorize all statements into suspicious group and unsuspicious group, with respect to the "*activated*" faults. The suspicious group should contain all the statements that have been demonstrated to have a chance of being faulty; while the unsuspicious group contains the remaining statements. Under such categorization, we only need to calculate the risks for suspicious statements, and simply to assign the risks of unsuspicious statements as the lowest value.

Even though some SBFL techniques may implicitly conceive that risk evaluation should avoid using the passed information alone; none of them has explicitly pointed it out, and actually some SBFL techniques do overlook this idea. Hence we explicitly summarize this simple but essential idea

into a basic rule, which should be considered in the further SBFL technique design.

More importantly, for the SBFL techniques that do not conceive such idea, our method can serve as an effective remedy. It is intuitively obvious that our method will not worsen the performance of these SBFL techniques, but how effective it can be is an interesting and important question. Thus we conducted an experimental study to investigate its effectiveness using the *Siemens suite* and 9 evaluation formulas. We discovered that our method have a good chance to significantly improve the performance of these formulas. Therefore, it is suggested to use this refinement method for these formulas, because it incurs minimal overhead but makes the predication more accurate, which will consequently reduce the debugging cost.

In our future study, we will complete the empirical study by using larger scale objective programs and multiple-faults mutants. Besides we will investigate the effectiveness of our method using some other schemes for ranking statements with the same risk values, such as average or the best-worst scheme. Additionally, how to improve R^{new} by distinguishing statements in G_u, with some additional information that can provide other definite or heuristic hints, will also be studied.

ACKNOWLEDGMENT

This project is partially supported by an Australian Research Council Discovery Grant (ARC DP0771733), the National Natural Science Foundation of China (90818027, 60633010, and 60721002), the National High Technology Development Program of China (2009AA01Z147), as well as the Major State Basic Research Development Program of China (2009CB320703).

REFERENCES

[1] M. Harrold, G. Rothermel, K. Sayre, R. Wu, and L. Yi, "An empirical investigation of the relationship between spectra differences and regression faults," *Software Testing Verification and Reliability*, vol. 10, no. 3, pp. 171–194, 2000.

[2] R. Abreu, P. Zoeteweij, and A. van Gemund, "On the Accuracy of Spectrum-based Fault Localization," in *Proceedings of Testing: Academic and Industrial Conference Practice and Research Techniques-MUTATION*, 2007, pp. 89–98.

[3] M. Chen, E. Kiciman, E. Fratkin, A. Fox, and E. Brewer, "Pinpoint: Problem determination in large, dynamic internet services," in *Proceedings of the 32th IEEE/IFIP International Conference on Dependable Systems and Networks*, 2002, pp. 595–604.

[4] J. Jones, M. Harrold, and J. Stasko, "Visualization of test information to assist fault localization," in *Proceedings of the 24th International Conference on Software Engineering*. ACM, 2002, p. 477.

[5] R. Abreu, P. Zoeteweij, and A. Gemund, "An evaluation of similarity coefficients for software fault localization," in *Proceedings of the 12th Pacific Rim International Symposium on Dependable Computing*. IEEE Computer Society, 2006, pp. 39–46.

[6] H. Pan and E. Spafford, "Heuristics for automatic localization of software faults," *World Wide Web*, 1992.

[7] T. Reps, T. Ball, M. Das, and J. Larus, "The use of program profiling for software maintenance with applications to the year 2000 problem," *ACM SIGSOFT Software Engineering Notes*, vol. 22, no. 6, pp. 432–449, 1997

[8] A. Zeller, "Isolating cause-effect chains from computer programs," *ACM SIGSOFT Software Engineering Notes*, vol. 27, no. 6, p. 10, 2002.

[9] B. Liblit, "Cooperative bug isolation," Ph.D. dissertation, University of California, 2004.

[10] B. Liblit, M. Naik, A. Zheng, A. Aiken, and M. Jordan, "Scalable statistical bug isolation," in *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*. ACM New York, NY, USA, 2005, pp. 15–26.

[11] C. Liu, X. Yan, L. Fei, J. Han, and S. Midkiff, "SOBER: statistical model-based bug localization," *ACM SIGSOFT Software Engineering Notes*, vol. 30, no. 5, pp. 286–295, 2005.

[12] W. Wong, Y. Qi, L. Zhao, and K. Cai, "Effective fault localization using code coverage," in *Proceedings of the 31st Annual International Computer Software and Applications Conference*, vol. 1, 2007.

[13] R. Hamming, "Error detecting and error correcting codes," *Bell System Technical Journal*, vol. 29, no. 2, pp. 147–160, 1950.

[14] L. Naish, H. J. Lee, and K. Ramamohanarao, "A model for spectra-based software diagnosis," *ACM Transactions on Software Engineering and Methodology*, in press.

[15] SIR, "http://sir.unl.edu/php/index.php."

[16] 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*. IEEE Computer Society Press Los Alamitos, CA, USA, 1994, pp. 191–200.

[17] M. Renieris and S. Reiss, "Fault localization with nearest neighbor queries," in *Proceedings of the 18th IEEE International Conference on Automated Software Engineering*, 2003, pp. 30–39.

[18] J. Jones and M. Harrold, "Empirical evaluation of the tarantula automatic fault-localization technique," in *Proceedings of the 20th IEEE/ACM International Conference on Automated Software Engineering*. ACM, 2005, p. 282.

[19] A. Jain and R. Dubes, *Algorithms for clustering data*. Prentice Hall Englewood Cliffs, NJ, 1988.

[20] M. Dunham, *Data Mining: Introductory And Advanced Topics, 1/e*. Pearson Education, 2003.

[21] W. Scott, "Reliability of content analysis: The case of nominal scale coding," *Public Opinion Quarterly*, vol. 19, no. 3, p. 321, 1955.

[22] A. Maxwell and A. Pilliner, "Deriving coefficients of reliability and agreement for ratings." *The British Journal of Mathematical and Statistical Psychology*, vol. 21, no. 1, p. 105, 1968.

[23] E. Rogot and I. Goldberg, "A proposed index for measuring agreement in test-retest studies." *Journal of chronic diseases*, vol. 19, no. 9, p. 991, 1966.

[24] B. Everitt, *Graphical techniques for multivariate data*. North-Holland New York, 1978.

[25] W. Wong, V. Debroy, and B. Choi, "A family of code coverage-based heuristics for effective fault localization," *The Journal of Systems and Software*, vol. 83, no. 2, pp. 188–208, 2010.

[26] X. Zhang, N. Gupta, and R. Gupta, "Locating faults through automated predicate switching," in *Proceedings of the 28th international conference on Software engineering*. ACM, 2006, pp. 281–290.

[27] D. Jeffrey, N. Gupta, and R. Gupta, "Fault localization using value replacement," in *Proceedings of the 2008 international symposium on Software testing and analysis*. ACM, 2008, pp. 167–178.

[28] W. Wong, T. Sugeta, Y. Qi, and J. Maldonado, "Smart debugging software architectural design in SDL," *The Journal of Systems and Software*, vol. 76, no. 1, pp. 15–28, 2005.