2011 11th International Conference On Quality Software

Spectrum-Based Fault Localization: Testing Oracles Are No Longer Mandatory

Xiaoyuan Xie[1, 3, 4 *], W. Eric Wong[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]Department of Computer Science
University of Texas at Dallas
Richardson, TX 75083, United States
ewong@utdallas.edu

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

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

Abstract—**Spectrum-based Fault Localization (SBFL) is one of the most popular approaches for locating software faults, and has received much attention because of its simplicity and effectiveness. It utilizes the execution result of each test case (*failure* or *pass*) and the corresponding coverage information to evaluate the likelihood of each program entity (e.g., a statement or a predicate) being faulty. Different formulas for computing such likelihood have been proposed based on different intuitions. All existing SBFL techniques have assumed the existence of a testing oracle, that is, a mechanism which can determine whether the execution of a test case *fails* or *passes*. However, such an assumption does not always hold. Recently, metamorphic testing has been proposed to alleviate the oracle problem. Thus, it is a natural extension to investigate how it can help SBFL techniques to locate faults even without using a testing oracle. Based on the framework of metamorphic testing, we have developed a novel concept of *mice* as a counterpart of the slice used in the current SBFL techniques. More precisely, in the absence of a testing oracle, we can determine whether an expected characterization of the program is satisfied. The outcomes of dissatisfaction or satisfaction of an expected characterization are then regarded as the counterparts of *failed* or *passed* executions, respectively, when a testing oracle exists. Since our approach does not require the existence of a testing oracle, it significantly enhances the applicability of SBFL techniques. Case studies on three popular SBFL techniques (Tarantula, Ochiai and Jaccard) with 9 applications are reported to demonstrate the use of the proposed fault localization technique.**

Keywords—spectrum-based fault localization, slice, mice, testing oracle, metamorphic testing

I. INTRODUCTION

It is commonly recognized that testing and debugging are important but resource consuming activities in software engineering. Attempts to reduce the number of delivered faults[1] in software are estimated to consume 50% to 80% of the total development and maintenance effort [1], in which, trying to locate the faults is one of the most essential but tedious tasks, due to a great amount of manual involvement. This makes fault localization a very resource consuming task in the whole software development life cycle. Therefore many researchers aim at proposing automatic and effective techniques for fault localization, in order to decrease its cost, as well as to increase the software reliability.

One promising approach for fault localization is Spectrum-Based Fault Localization (referred to as SBFL in this paper). Generally speaking, this dynamic approach utilizes various program spectra and their associated testing results, in terms of *failure* or *pass*, for each test case. The program spectrum can be any granularity of program entities. For example, one of the most widely adopted spectra is the execution slice recording the coverage on statement level for a certain test case [2].

After collecting all the necessary information, SBFL uses different formulas to evaluate the likelihood of containing a fault for each program entity, and gives a likelihood ranking list. SBFL intends to highlight program entities which strongly correlate with program failures, and which are regarded as the likely faulty locations [3]. Some typical SBFL techniques include Jaccard [4], Tarantula [5], Ochiai [6], etc [7-14].

SBFL has received a lot of attention due to its simplicity and effectiveness. However all existing SBFL techniques have assumed the existence of a testing oracle. But such an assumption does not always hold in many programs, such as complex computational programs, machine learning algorithms, etc [15-20]. This prerequisite has made SBFL infeasible in many application domains.

Recently, metamorphic testing has been proposed to alleviate the oracle problem. It is a natural extension to investigate how it can help SBFL techniques to locate faults when there is no testing oracle. Thus in this paper, within the framework of metamorphic testing, we propose a novel concept of *mice* as a counterpart of the slice used in the current SBFL techniques. More precisely, this concept is property-based, that is, it is related to certain expected characterization of the program under testing. Even in the absence of a testing oracle, the outcomes of dissatisfaction or satisfaction of this property can still be determined, which is denoted as *violated* or *non-violated*, and is regarded as the counterparts of *failed* or *passed* executions, respectively, for the situation that a testing oracle exists. With *mice*, testing oracles are no longer mandatory in SBFL; hence our method significantly enhances the applicability of the current SBFL techniques.

In our case study, we investigate 9 programs in different sizes, which have the oracle problem with respect to some of their functionalities. Using three widely studied SBFL techniques, we investigate the applicability of our method. The empirical results reveal that our method is applicable for these programs where there are no testing oracles and hence the

* Corresponding author

[1]We use “fault” and “bug” interchangeably.

conventional SBFL cannot really be adopted in practice. Moreover, using our method almost has no loss on effectiveness, compared to the performance of the conventional SBFL evaluated with the assistance of a "assumed oracle". Hence, our method can make the testing oracle no longer mandatory in SBFL.

The rest of this paper is organized as follows. Section II introduces the background of spectrum-based fault localization and metamorphic testing. Section III gives the concept of *mice*, and also describes how it is used in SBFL to alleviate the oracle problem. In Section IV, we present the experimental setup for our case study. We report and analyze the empirical results in Section V. Section VI describes the conclusion and future work.

II. Background

A. *Spectrum-based fault localization (SBFL)*

During software testing, two essential types of information are collected for SBFL, namely program spectrum and testing results.

A program spectrum is a collection of data that provides a specific view on the dynamic behavior of software [11]. Generally speaking, it records the run-time profiles about various program entities for a specific test suite. The program entities 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 entity has been covered, the program state before and after executing the program entity, etc. In practice, there are many kinds of combinations [21]. The most widely adopted combination involves statement and its coverage status in one test execution, which is known as execution slice (referred to as *e_slice* in this paper) [2]. In the following discussion, we will focus on this *e_slice* as a representative type of spectrum adopted by the conventional SBFL.

Apart from the program spectrum, the testing result associated with each test case is also essential to SBFL. It records whether a test case is *failed* or *passed*. Together with the coverage information, the testing results give debuggers hints about which statements are more likely to contain the faults.

Given a program P with n statements and executed by m test cases, Fig. 1 summarizes the essential information required by SBFL. Vector TS contains the m test cases and matrix M^{TS} represents the program spectrum, in particular, the *e_slice* for each test case. The element in the i^{th} row and j^{th} column of matrix M^{TS} represents the coverage information of statement s_j, by the test case t_i, with 1 indicating s_j is executed, and 0 otherwise. Vector R^{TS} records the testing result of *failure* or *pass* for each individual test case.

$$P:(s_1 \quad s_2 \quad \ldots \quad s_n)$$

$$TS:\begin{pmatrix} t_1 \\ t_2 \\ \cdot \\ \cdot \\ \cdot \\ t_m \end{pmatrix} \quad M^{TS}:\begin{pmatrix} 1/0 & 1/0 & \ldots & 1/0 \\ 1/0 & 1/0 & \ldots & 1/0 \\ & \cdot & & \\ & \cdot & & \\ & \cdot & & \\ 1/0 & 1/0 & \ldots & 1/0 \end{pmatrix} \quad R^{TS}:\begin{pmatrix} failed/passed \\ failed/passed \\ \cdot \\ \cdot \\ \cdot \\ failed/passed \end{pmatrix}$$

Figure 1. Essential information for conventional SBFL

For each statement s_j, these data can be reformulated into a vector of four elements, denoted as $A^j = < a^j_{ef}, a^j_{ep}, a^j_{nf}, a^j_{np} >$, where a^j_{ef} and a^j_{ep} represent the number of test cases that execute the relevant statement s_j and return the *failed* and *passed* testing results, respectively. While a^j_{nf} and a^j_{np} stand for the number of test cases that do not execute s_j, and return the *failed* and *passed* testing results, respectively. A risk evaluation formula f can be used to map the vector $A^j = < a^j_{ef}, a^j_{ep}, a^j_{nf}, a^j_{np} >$ to a risk value r_j. After collecting the risks for all statements, a ranking list for the risks is compiled in descending order. Debuggers are supposed to inspect the statements according to this ranking list from top to bottom.

Existing SBFL techniques include Jaccard [4], Ochiai [6], Tarantula [5], Wong [12], etc. They have been studied because they are simple in concept, easily to apply, and furthermore, they are effective as demonstrated by the extensive experimental analysis. However, there are still some problems in SBFL, and one of the most crucial problems is the assumption of the existence of testing oracle.

Actually in real-world applications, there are many programs having the oracle problem, that is, it is impossible or too expensive to verify the correctness of the computed outputs. For example, when testing a compiler, it is not easy to verify whether the generated object code is equivalent to the source code. Other examples include testing programs involving machine learning algorithms, simulations, combinatorial calculations, graphics display on the monitor, etc [15-20]. In such cases, current SBFL techniques cannot be applied.

B. *Metamorphic testing*

Metamorphic testing (MT) [22-27] has been recently proposed to alleviate the oracle problem. Instead of verifying the correctness of the computed outputs of individual test cases, MT uses some specific properties of problem domain, namely metamorphic relations (MRs), to verify the relationship between multiple but related test cases and their outputs.

Generally speaking, when conducting MT, the testers first need to identify an MR of the software under test. It is obvious that an MR always involves multiple test cases, in which some are designated as the source test cases whose generation is independent to the identification of MR and can be obtained according to various selection strategies, such as random testing, coverage criteria-based testing, etc. The other test cases are then constructed from the source cases and their outputs according to the MR, and are referred as the follow-up test cases. For convenience of reference, we will refer to a source test case (or a group of source test cases) and its related follow-up test cases as a metamorphic test group. Then, the program is executed with all test cases of a metamorphic test group, and their corresponding outputs are recorded. However, instead of verifying the correctness of the output of each individual test case, the MR is verified with respect to the metamorphic test group and its outputs. *Violation* of MR implies an incorrect implementation.

Let us use an example to illustrate the process of MT. Our example is about a program that searches for the shortest path between any two nodes in an undirected graph and reports the length of the shortest path. Let us denote the length of the shortest path by $d(x, y, G)$, where x is the start node, y is the destination node and G is the graph. Suppose that the computed value of $d(x, y, G)$ is 13579. It is very expensive to check whether 13579 is correct due to the combinatorially large number of possible paths between x and y. Therefore, such a program is said to have the oracle problem. When applying MT to this program, we first need to define an MR based on some well-known properties in graph theory. For example, one possible MR (referred as MR1) is that the length of the shortest path will remain unchanged if we swap the start node and destination node, that is, $d(x, y, G) = d(y, x, G)$. Another possible MR (referred as MR2) is that suppose w is a node in the shortest path with x as the start node and y as the destination node, then $d(x, y, G) = d(x, w, G) + d(w, y, G)$. The idea is that although it is difficult to verify the correctness of the individual output, namely $d(x, y, G)$, $d(y, x, G)$, $d(x, w, G)$ and $d(w, y, G)$), but it is easy to verify whether the MR1 and MR2 are satisfied or not, that is, whether $d(x, y, G) = d(y, x, G)$ and $d(x, y, G) = d(x, w, G) + d(w, y, G)$. In other words, we can run the program using y as the start node and x as the destination node. If $d(y, x, G)$ is not equal to 13579, then we can conclude that the program is incorrect. However, if $d(y, x, G)$ is also 13579, we can neither conclude the program is correct, nor incorrect. But, this is the limitation of software testing. In this example, (x, y, G) is referred as the source test case, and (y, x, G) is the follow-up test case of MR1, and (x, w, G) and (w, y, G) are the follow-up test case of MR2. It can be seen that there could be multiple follow-up test cases and that follow-up test cases are not only dependent on the source test cases but also on their outputs. As a reminder, the source test case need not be a single test case and it can be selected according to any test case selection strategies.

It should be obvious that metamorphic testing is simple in concept, easily automatable, and independent of any particular programming language. Nevertheless, selection of effective MRs may not be straightforward. But this problem is beyond the scope of this paper. Interested readers may consult [28].

III. Approach

In order to make existing SBFL techniques feasible for the application domains without testing oracles, in Section III.A we first define a new concept, namely, "*mice*", which is a group of slices bound together with certain property of the algorithm under investigation. With *mice*, we are able to extend the application of the SBFL techniques beyond the programs that must have testing oracles, in Section III.B.

A. Mice

In the family of program slices, there are three main types of slices, namely static slice, dynamic slice, and execution slice [2, 29-31]. Dynamic slice and execution slice are normally used in SBFL, whose definitions are as follows:

- Given a variable v and a test case t, a dynamic slice denoted as *d_slice(v, t)*, is the set of statements that have actually affected the variable in the test run with t.
- Given a test case t, an execution slice denoted as *e_slice(t)*, is the set of statements that have been covered in test run with t.

As mentioned above, when using either of these two slices in SBFL, we must always identify the testing outcome of *failure* or *pass* associated with each individual *d_slice* or *e_slice* for test case t. However in reality, such identification is impossible if the program under testing has the oracle problem. Recently metamorphic testing has been proposed to alleviate the oracle problem. Thus, it is natural to consider how to integrate the concepts of the slices with metamorphic testing, to extend the application of SBFL to those areas where the testing oracle does not exist. Therefore, we propose a new concept of *mice* to achieve this goal.

Intuitively speaking, a *mice* is the group of slices which are bound together with certain program property (also known as MR). In particular, corresponding to *d_slice* and *e_slice*, there are dynamic *mice* and execution *mice* as following:

For a given a metamorphic relation *MR,* a set of *ks* source test cases $T_i^S = \{t_1^S, \ldots, t_{ks}^S\}$ required by *MR*, the corresponding set of *kf* follow-up test cases $T_i^F = \{t_1^F, \ldots, t_{kf}^F\}$ decided by T^S_i and *MR*, and a variable v, we have

- The dynamic *mice*, *d_mice(v, MR, T^S_i)* is the union of all *d_slice*(v, t) for t that have been involved in the relevant MT executions with T^S_i and T^F_i, that is,
$$d\_mice(v, MR, T_i^S) = (\bigcup_{k=1}^{ks} d\_slice(v, t_k^S)) \cup (\bigcup_{k=1}^{kf} d\_slice(v, t_k^F))$$
- The execution *mice*, *e_mice(MR, T^S_i)* is the union of all *e_slice*(t) for t that have been involved in the relevant MT executions with T^S_i and T^F_i, that is,
$$e\_mice(MR, T_i^S) = (\bigcup_{k=1}^{ks} e\_slice(t_k^S)) \cup (\bigcup_{k=1}^{kf} e\_slice(t_k^F))$$

Essentially, both the *d_mice(v, MR, T^S_i)* and *e_mice(MR, T^S_i)* are sets of program statements.

Technically speaking, a *mice* has actually bound all the relevant *d_slice*(v, t) (or *e_slice*(t)) of test cases t belonging to a metamorphic test group of *MR*. According to metamorphic testing, with a given MR, each *d_mice* or *e_mice* must be associated with a metamorphic testing result of *violation* or *non-violation*. More importantly, identifying such result can always be carried out even there is no available testing oracle. In other words, regardless of the availability of the testing result associated with each individual *d_slice* or *e_slice*, a metamorphic testing result of *violation* or *non-violation* associated with each *d_mice* or *e_mice*, is always available.

Consequently with such integration of slice and metamorphic testing, the testing oracles are no longer mandatory for SBFL. Thus we are able to extend the application of SBFL beyond the programs that must have testing oracles. Corresponding to the SBFL using *e_slice* introduced in Section II.A, we will describe in detail how *e_mice* can serve as an alternative of *e_slice*, to alleviate the oracle problem in SBFL in Section III.B.

B. SBFL with e_mice

Without loss of generality, we assume that the given metamorphic relation *MR* has only one source test case and one follow-up test case, that is, T^S_i and T^F_i have one and only one element each. Let us use g_i to denote the corresponding metamorphic test group for T^S_i and T^F_i.

Obviously, in SBFL with *e_mice*, the coverage information is provided by the *e_mice* of each g_i. According to the definition, in each g_i, the *e_slice* of both T^S_i and T^F_i are collected and their union is recorded as the corresponding *e_mice* for g_i. Since *e_mice* is bound to *MR*, after checking the relation between the outputs of T^S_i and T^F_i against *MR*, a metamorphic testing result of being *violated* or *non-violated* would be associated with each g_i.

Suppose *m* T^S_i are used as the source test suite, then there would be *m* T^F_i constructed from it based on *MR*, and *m* metamorphic test groups g_i. The program should be executed with both T^S_i and T^F_i of each g_i, that is, for 2**m* times. Consequently there should be *m e_mices* and *m* corresponding MT results in total. With this information, we can construct the similar matrix and vectors in Fig. 2, as the conventional SBFL.

$$P : (s_1 \quad s_2 \quad \ldots \quad s_n)$$

$$MTS : \begin{pmatrix} g_1 \\ g_2 \\ \cdot \\ \cdot \\ \cdot \\ g_m \end{pmatrix} \quad M^{MTS} : \begin{pmatrix} 1/0 & 1/0 & \ldots & 1/0 \\ 1/0 & 1/0 & \ldots & 1/0 \\ & & \cdot & \\ & & \cdot & \\ & & \cdot & \\ 1/0 & 1/0 & \ldots & 1/0 \end{pmatrix} \quad R^{MTS} : \begin{pmatrix} vio/non\_vio \\ vio/non\_vio \\ \cdot \\ \cdot \\ \cdot \\ vio/non\_vio \end{pmatrix}$$

Figure 2. Essential information for SBFL with *e_mice*

In Fig. 2, the vector *MTS* is the test suite containing *m* metamorphic test groups. Matrix M^{MTS} represents the program spectrum constructed using *e_mice*, and in each of its row sub-vectors, the binary value of 1 denotes the membership of the corresponding statement in *e_mice(MR,* T^S_i*)*, and 0 otherwise. Each row in vector R^{MTS} records the corresponding metamorphic testing result for each g_i, indicating *violated* or *non-violated*.

It is not difficult to see that in extending the SBFL to programs without testing oracles, the individual test case t_i is replaced by the metamorphic test group g_i; the *e_slice* for each t_i is replaced by *e_mice* of a specific *MR* for each g_i; and the testing result of *failed* or *passed* is replaced by the metamorphic testing result of *violated* or *non-violated*. After such replacements, the same procedure can be applied to reformulate the vector $A^j = < a^j_{ef}, a^j_{ep}, a^j_{nf}, a^j_{np} >$ and then evaluate the risk value r_j for each statement s_j.

Actually, SBFL using *e_mice* has similar intuition as SBFL using *e_slice*. That is, a *failed* test case implies that a faulty statement is included in the corresponding *e_slice*, while a *passed* test case does not provide a definite conclusion whether the corresponding *e_slice* is free of faulty statement. Similarly, a *violated* metamorphic test group implies that there is at least one *failed* test case within it. Even though we do not know which test cases are actually the *failed* ones, we still can conclude that a faulty statement must be included in the union of all the corresponding *e_slices,* that is, the *e_mice*. On the other hand, a *non-violated* metamorphic test group does not provide a definite conclusion about whether it involves any *failed* test case and as a consequence, the correctness of all statements in the current *e_mice* is also uncertain.

IV. EXPERIMENTAL SETUP

A. *Testing objectives*

In the investigation of the effectiveness of our fault-localization method using *e_mice*, we have selected the ***Siemens Suite***, as well as two other programs, namely ***SeqMap*** and ***grep***, as our testing objectives.

The ***Siemens Suite*** includes 7 programs in C, with some details listed in Table 1. The versions of all the 7 programs used in our experiments are 2.0, which are downloaded from Software-artifact Infrastructure Repository (SIR) [32].

TABLE I. BASIC INFORMATION OF SIEMENS SUITE

Program	**Mutant Versions**	**Number of Test Cases**	**Description**
print_tokens	7	4130	Lexical analyzer
print_tokens2	10	4115	Lexical analyzer
replace	32	5542	Pattern recognition
schedule	9	2650	Priority scheduler
schedule2	10	2710	Priority scheduler
tcas	41	1608	Altitude separation
tot_info	23	1052	Information measure

SeqMap is a bioinformatics tool in C++, for mapping massive amount of short sequence reads to a reference genome by solving an approximate string matching problem [33]. One of its important applications is the detection of cross-hybridizing probes in an oligonucleotide microarray [16, 34]. The version in our experiments is 1.0.8 obtained from [35].

And ***grep*** is a UNIX command-line utility program written in C, which performs the pattern matching. The version in our experiments is version 1.2 from SIR website [32].

In our experiments, we have chosen the ***Siemens*** programs, because, though small in sizes, they were extensively used in previous SBFL studies. And we also include ***SeqMap*** and ***grep*** because they are larger in size than the ***Siemens*** programs.

Apart from the above reason, more importantly, checking the correctness of the outputs for these programs is actually difficult in most cases, that is, they have the oracle problem. For example, in ***SeqMap***, although it is easy to verify in the output whether the listed mapping information is correct or not, it is very difficult to check whether all the possible mapping positions have been printed [16]. In other words, soundness of the results may be easy to verify, but not the completeness of the results. As a reminder, all the previous SBFL studies simply assume the original versions as the testing oracles.

Furthermore, in our experiments, we only consider the executable program lines in each testing objective, that is, we exclude those statements such as blank lines, lines of left or right brace, etc. Table II lists the numbers of the executable lines for all the 9 programs, collected by a tool called SLOCCount (version 2.26) which is downloaded from [36].

TABLE II. EXECUTABLE LINE NUMBERS OF EACH PROGRAM

Program	**Exe_LOC**
print_tokens	342
print_tokens2	355
replace	512
schedule	292
schedule2	262
tcas	135
tot_info	273
SeqMap	1783
grep	7309

All experiments are conducted on a cluster of 64-bit Intel Clovertown systems running CentOS 5. The statement coverage is collected by using ***gcov***.

B. Definition of MR

Since *e_mice* is an MR-related concept, we first define metamorphic relations for each program. In this case study, we enumerate three MRs for each program. Due to the limitation of space, we only describe the MRs for ***grep*** in this paper, whose specification is precisely and completely described in its manual that can be easily obtained from the GNU website [37].

For ***grep*** program, our MRs are related to the regular expression analyzer, which is one of its most important components. With an input file "*file.in*" and a given input regular expression RE_s, ***grep*** scans the "*file.in*" line by line, searching for strings that can match RE_s, and then prints all the searched information according to the specified options in command lines. MRs are used to generate follow-up regular expression RE_f which is equivalent to RE_s, meanwhile keeping the input file and the options unchanged. Since RE_s and RE_f are equivalent but different in format, their corresponding outputs O_s and O_f should be exactly the same as each other. The detailed definition of each MR is as follows:

MR1: Completely de-compositing the brackets structure

According to the specification of ***grep***, the regular expression within square brackets like "[*x*–*y*]" where *x* and *y* are digits and $x<y$, should match any digit within the range of [*x*, *y*]. One of its equivalent forms is a complete decomposition of this "[ ]" structure using "|". For example: "[1–3]" is equivalent to "1|2|3" or "2|1|3". Hence in MR1, RE_f is constructed through replacing such bracketed substrings in RE_s with their corresponding decomposed equivalent forms. This change should not produce any changes in the outputs.

MR2: Splitting the brackets structure

MR2 is also on the sub-strings within square brackets, but it does not completely decompose it. Instead, it splits this bracket structure into two parts using "|", and each part still keeps the bracket structure. For example, "[1–6]" can be equivalently split into "[1–3] | [4–6]"; while "[*abcd*]" can be transformed into "[*ab*]|[*cd*]". In this MR, RE_f is constructed by replacing a bracketed substring with its equivalent splits. Obviously we have $O_s = O_f$.

MR3: Bracketing simple characters

Based on the specification of ***grep***, apart from those key words with special meanings, any simple character should be equivalent to itself enclosed by the brackets, that is, "*a*" is equivalent to "[*a*]". Hence MR3 randomly selects a collection of simple characters in RE_s, and then encloses them one by one with brackets to form the follow-up regular expression RE_f. For example, the "*abc*" in RE_s can be replaced by "[*a*][*b*][*c*]" in RE_f. Again, the relation $O_s = O_f$ should be held.

C. Test suite generation

Normally, in MT, a test suite generated based on any test case selection strategy can be used as the source test suite, from which, the follow-up test suite can then be constructed according to the MR. In our experiments, we adopt the test suite provided by SIR in Table I as the source test suite, for each program in ***Siemens Suite***.

While for ***SeqMap***, since there is no existing test suite, we use the test suite of 300 random test cases that were used in our previous study as source test cases [16].

As for ***grep***, even though the SIR has 807 test cases, we do not utilize them because only quite a small portion of them are "eligible source test cases" with respect to our MR definitions. In other words, most of these test cases do not contain specific sub-strings in their regular expressions where our MRs are applicable. Consequently no equivalent transformation can be conducted. Therefore, in order to have sufficient source test cases for our experiments, we use a test pool with 171,634 random test cases, which was generated and used in another of our studies. Among them, 2982 test cases are eligible for MR1, 5003 for MR2, and 2084 for MR3.

As a reminder, the generation of each source test suite is totally independent to the identification of the MRs.

D. Mutant generation

Similar to the situations in test case generation, SIR has provided several released mutants for ***Siemens Suite*** and ***grep***. However, the amounts of the mutants are small for our study, since many of them are actually equivalent mutants with respect to our selected MRs, and on which no violation can be revealed. It is commonly adopted that, mutants with no failure revealed were excluded in the existing studies of SBFL with *e_slice*. Likewise, mutants with no violation are also excluded in our investigation. Consequently the eligible mutants are even fewer for our analysis. Therefore, for the ***Siemens*** and ***grep*** programs, apart from the provided mutants, we randomly generate 300 extra mutants. For ***SeqMap***, since there is no existing mutant available, we directly generate 300 mutants randomly.

Our mutant generation focuses on the non-omission and first-order faults (that is, each mutant contains exactly one bug. Investigation on multiple-faults will be conducted in our future study. Two types of mutant operators are used: statement mutation and operator mutation. For statement mutation, either a *continue* statement is replaced with a *break* statement (and vice versa), or the label of a *goto* statement is replaced with another valid label. For operator mutation, it involves the substitution of an *arithmetic* (or a *logical*) operator by a different *arithmetic* (or *logical*) operator. To generate a mutant, our tool first randomly selects a line in the relevant part within the source code, and then searches systematically for possible locations where a mutant operator can be applied. One of these mutant operators is then selected randomly and applied to one of the corresponding possible locations, which is also randomly selected.

For each program, the mutants which could not be compiled successfully are first excluded. And the ones with an exceptional exit (crash-failure) that prevent the coverage information from being correctly collected by ***gcov*** are also excluded. Of course, mutants without any *violated* metamorphic test group are excluded as well. As a result, the numbers of the available mutants for each program in each MR are shown in Table III.

TABLE III. NUMBER OF AVAILABLE MUTANTS

Program	MR1	MR2	MR3
print_tokens	24	32	21
print_tokens2	31	27	21
replace	49	53	42
schedule	31	27	40
schedule2	36	37	21
tcas	16	26	27
tot_info	21	27	28
SeqMap	49	27	21
grep	78	32	36

E. *Three risk evaluation formulas*

In our study, we focus on three SBFL techniques, namely Tarantula [5], Ochiai [6] and Jaccard [4], whose risk evaluation formulas are listed in Table IV (the notations in these formulas have been described in Section II.A). Our experiments compare the effectiveness (to be explained in the following paragraph) of using *e_mice* and *e_slice* with these formulas and these formulas will be referred as the name of their original techniques in our experiments. .

TABLE IV. RISK EVALUATION FORMULAS

Technique Name	Formula
Tarantula	$\frac{a_{ef}}{a_{ef}+a_{nf}} / (\frac{a_{ef}}{a_{ef}+a_{nf}} + \frac{a_{ep}}{a_{ep}+a_{np}})$
Ochiai	$a_{ef} / \sqrt{(a_{ef}+a_{nf})(a_{ef}+a_{ep})}$
Jaccard	$a_{ef} / (a_{ef}+a_{nf}+a_{ep})$

Following other fault localization studies [7, 9, 12-14, 38], the effectiveness of a fault localization technique can be measured in terms of how much code has to be examined (or not examined) before the first faulty statement is identified. While the authors of [38] define a score in terms of the percentage of code that *need not be* examined in order to find a fault, we feel it is more intuitively appealing to use the percentage of code that *has to be examined* in order to find the fault. This modified score (referred to as the *EXAM* score [12, 13]) is adopted in our study and is defined as the percentage of executable statements that have to be examined until the first statement containing the bug is reached. For statements with the same risk values, we rank them according to their original order in the source code rather than use the *best* and *worst* scenarios as discussed in studies such as [12, 13].

V. EXPERIMENTAL RESULTS AND ANALYSIS

For each program, we applied SBFL with *e_mice* using all relevant MRs and each formula in Table IV, aiming to investigate whether our proposed method is effective to locate faults without referring to a testing oracle. In our experiments we refer to the performance of the conventional SBFL with *e_slice*, which is evaluated with the assistance of an “assumed oracle”, as the benchmarks. Therefore we have conducted SBFL under the following four scenarios:

- MS: SBFL using *e_mice* with all eligible metamorphic test groups
- S-ST: SBFL using *e_slice* with all eligible source test cases
- S-FT: SBFL using *e_slice* with all eligible follow-up test cases
- S-AT: SBFL using *e_slice* with all eligible source and eligible follow-up test cases

As a reminder, we do such comparison because the numbers of *e_slices* and *e_mices* used in S-ST, S-FT and MS are the same, while S-AT has used twice as many *e_slices*. In other words, S-ST, S-FT and MS have the same amount of raw data for the evaluation formulas, and hence delivery the same degree of reliability of the prediction data. MS and S-AT have the same number of test executions, which is twice the number of test executions of S-ST and S-FT. Thus the program execution overheads of MS and S-AT are the same, while the program execution of S-ST and S-FT are also similar, but about half of those for either MS or S-AT.

As mentioned above, in S-ST, S-FT and S-AT, we use an “assumed oracle”, which is the non-mutated original version of program, to determine whether the execution of a mutant on a given test case is *failed* or *passed*. While for MS, since *e_mices* are used, such an “assumed oracle” is not required, instead, the *violated* or *non-violated* information to each *e_mice* for each metamorphic test group is used.

In the following sections, we analyze the experimental results based on *EXAM* distribution and average *EXAM* score for each program.

A. *EXAM distribution*

To demonstrate the applicability of our method MS, we first compare its *EXAM* distribution among all the mutants of a program using a particular formula, with the other three methods S-ST, S-FT and S-AT. Due to the space limitation, we do not present the diagrams for all the 9 programs; rather, we select one of the larger programs, ***grep***, as well as two smaller programs, ***print_tokens2*** and ***tot_info*** *from* ***Siemens Suite*** as representatives. The *EXAM* distributions for each of these programs are presented in Fig. 3 to Fig. 5. Each figure collects and summarizes the results of all the three MRs, with formulas Tarantula, Ochiai and Jaccard respectively, for a specific program.

In these figures, the horizontal axis means the *EXAM* score, while the vertical axis means the cumulative percentage of the mutants whose *EXAM* scores are less than or equal to the corresponding *EXAM* score. Obviously, the faster that the graph increases and reaches 100% of mutants, the better performance of its corresponding method is.

As seen in Fig. 3, for the relatively larger program ***grep***, with all the three formulas, the curve of MS is always above the other three methods, until they all achieve 100% of mutants under the *EXAM* scores around 30%. This indicates that MS has more mutants with lower *EXAM* scores than the other three, that is, MS performs better. A closer inspection shows that with all the three formulas, MS actually has a significant portion of mutants with *EXAM* scores less than 5%. With the increase of *EXAM*, the percentage of mutants accumulates sharply, and achieves the 100% very quickly. For example, referring to formula Ochiai, MS has over 45.2% of the mutants with *EXAM* lower than 5%. And at the *EXAM* score of 10%, there are over 89.7% of the mutants. Finally, the curve has reached the top point before *EXAM* scores reaching 15%, which means there are no mutant whose *EXAM* is greater than 15%.

In fact, similar results are observed for program ***SeqMap***. Especially with formula Tarantula, over 97.9% of the mutants have *EXAM* lower than 40%, while at the same *EXAM* score of 40%, S-ST, S-FT and S-AT have 61.8%, 63.9% and 63.9% of the mutants, respectively. On the other hand, even though the difference in performance is less considerable in the other two formulas, MS still always performs better than S-ST.

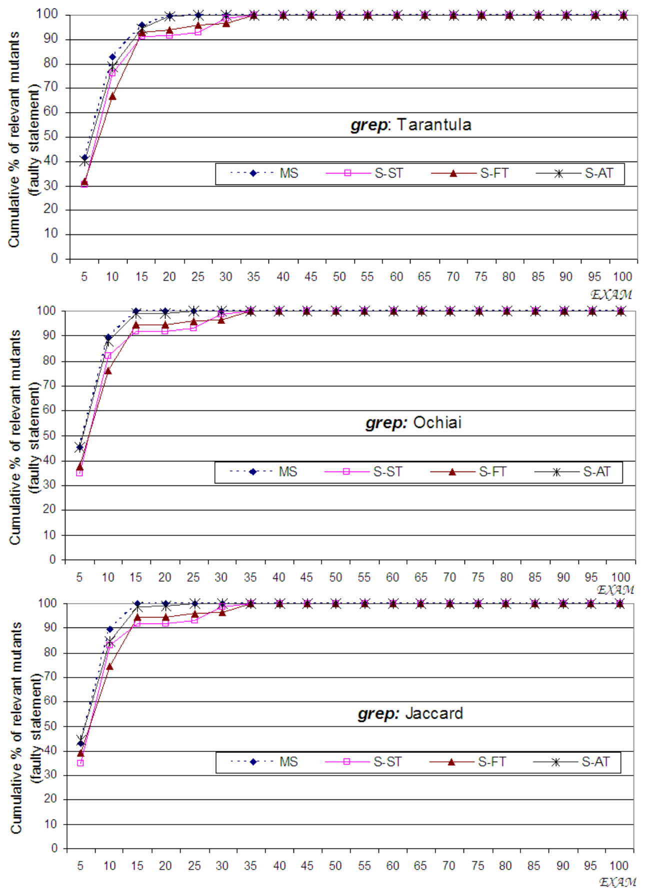

Figure 3. *EXAM* distribution in ***grep***

More importantly, we also find that using some MRs, there exist mutants producing no failure with the source test suite, but revealing violations in the metamorphic executions. This reveals that *mice* are not only quite helpful in alleviating the oracle problem of conventional SBFL using slice, but also able to make the original "useless" test suite that without any failed test case become useful through revealing the violations.

On the other hand, in the 7 smaller programs from ***Siemens Suite***, the performance of MS varies among these 7 programs with different formulas. For example in program ***schedule2***, the results are similar to the ones in the ***grep*** and ***SeqMap*** programs, that is, MS has better or similar performance as the other three methods.

However in program ***print_tokens2*** and ***replace***, MS only has a similar performance as the other three methods. Due to the space, we only present the *EXAM* distribution for program ***print_tokens2*** in Fig. 4 as an illustration.

It can be seen in Fig. 4 that the curves of MS for program ***print_tokens2*** are inter-crossing with the other three curves. For example, MS may perform slightly worse at the low *EXAM* score, but its curves increase sharply with the increase of the *EXAM* scores, and finally exceed the other three curves to reach 100% of mutants at a smaller *EXAM* score. Actually in program ***replace***, these four methods have almost identical performance as shown by the almost identical curves of MS, S-ST, S-FT and S-AT.

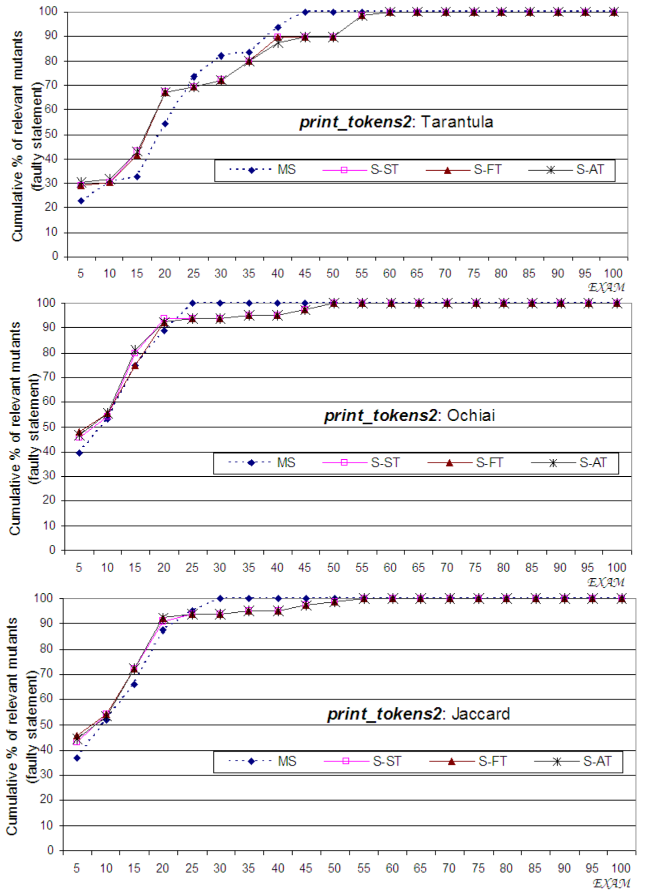

Figure 4. *EXAM* distribution in ***print_tokens2***

For the remaining programs, ***print_tokens***, ***schedule***, ***tcas*** and ***tot_info***, MS performs similarly as the other three methods using Tarantula formula, but worse using formulas Ochiai and Jaccard. As an illustration, referring to the *EXAM* distribution for program ***tot_info*** shown in Fig. 5, it can be seen that when using formula Tarantula, the performance of MS is worse than the other three methods for *EXAM* score lower than 10%, but becomes better when the *EXAM* score is greater than 10%. For the other two formulas, the curves of MS are always underneath those for S-ST, S-FT and S-AT. However, even in these worse situations, the increasing trend of the MS is quite satisfactory. It can be seen that with either formula Ochiai or Jaccard, MS has only 30% of mutants having smaller than 5% *EXAM* scores, and has nearly 90% of mutants having smaller than 15% *EXAM* scores. However, MS almost catches up with the other three methods at the *EXAM* score of 25%-30%.

From the above experimental results, it can be observed that as compared with S-ST, S-FT and S-AT, MS may perform either better, similarly, or worse. Therefore for each program, we further analyze the average *EXAM* score, to investigate the difference of the average *EXAM* score among MS, S-ST, S-FT and S-AT.

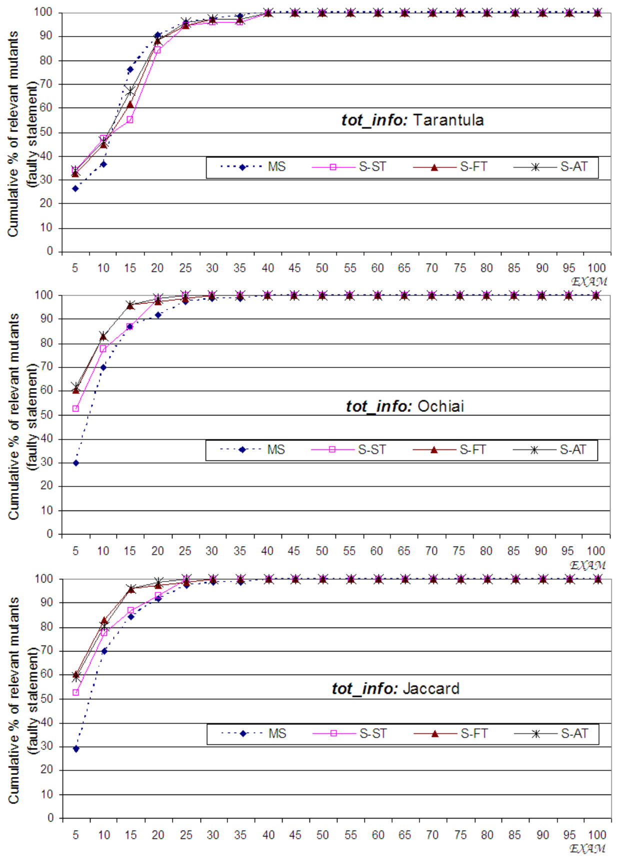

Figure 5. *EXAM* distribution in ***tot_info***

B. Average *EXAM* score

In our experiments we have summarized the average *EXAM* score among all mutants of all MRs, in each program with each formula for MS, S-ST, S-FT and S-AT, respectively. But due to the space, we only present the results for the three selected programs in Section V.A, in Fig. 6.

For programs ***grep*** and ***print_tokens2***, the results are consistent with the ones in *EXAM* distribution analysis. In ***grep***, the average *EXAM* scores of MS is always no larger than the other three methods. And in ***print_tokens2***, the performances of these four methods are quite similar to each other, with all the three formulas. Besides, from the average *EXAM* scores comparison, we can find that MS actually can have slightly smaller *EXAM* scores than the other three methods, with all the three formulas.

While in program ***tot_info***, though the average *EXAM* of MS is higher than those of the other three methods, this observation is consistent with the analysis using the *EXAM* distribution, but the difference among their average *EXAM* score is actually very small. For other programs, the analysis outcomes using average *EXAM* sores are consistent with the ones using the *EXAM* distribution, in particular the differences between average *EXAM* scores are small.

From the investigation on average *EXAM* scores, we can find that even for cases under which MS performs worse than the other three methods, our method is still worthwhile to apply, because the performance difference is actually small and MS may have violated metamorphic testing groups even when S-ST has passed outcomes for the corresponding source test cases. More importantly, we should always remember that MS can make SBFL applicable for programs without testing oracle, and that it is common to have no testing oracle in practice.

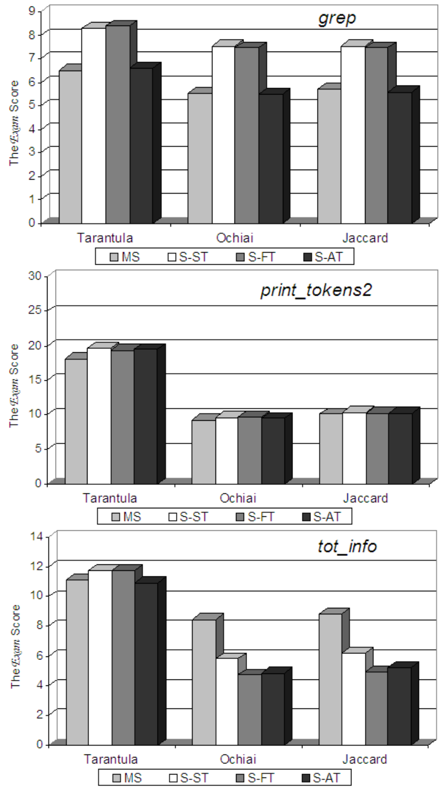

Figure 6. Average *EXAM* scores

C. Summary

According to the analysis from the above two aspects, we can conclude that the performance of our method is comparable to the conventional SBFL using *e_slice*, in terms of the *EXAM* score.

It should be noted that our primary objective is to investigate how to extend the applicability of SBFL to programs without testing oracles; thus, the effectiveness of our proposed method is not the main concern. Nevertheless, it is still very encouraging to observe that our method can perform similarly or better than the conventional SBFL using an "assumed oracle" in some cases. And even in other cases where our method performs worse than the conventional SBFL, the difference in performance is still small. Therefore, our method has successfully extended SBFL to applications where no testing oracle exists.

D. *Threats to Validity*

1) *External validity*

The primary threat to the external validity is the representative of our results acquired from the 9 testing objectives, with only three MRs for each. It is known that the 7 programs in ***Siemens Suite*** are small in size. Although ***grep*** and ***SeqMap*** is considerably larger than these 7 programs, they are still not very large-sized programs. Thus it is worthwhile to use more large-sized programs to further validate the effectiveness of our method of using *e_mice* for locating software faults.

Another threat is the type of mutants (namely, the type of faults) used in our study. Even though these mutants are randomly generated, each one, however, only contains exactly one fault and the types of faults are also limited. However, our approach can be extended to programs with multiple faults. When two or more test cases result in a program execution which does not satisfy certain expected characterization, it is not necessary that all the dissatisfaction is caused by the same fault. However, if there is a way (e.g., the fault focused approach in [39]) to segregate or rather cluster dissatisfied executions together such that violated test groups in each cluster are related to the same fault, then these violated test groups along with some non-violated test groups can be used to localize the corresponding causative fault. As a result, debugging a program with multi faults can be decomposed as debugging a set of programs each with a single fault.

We understand that identifying effective MRs may be difficult for some programs. Though in this paper such identification has been easily done, without a more comprehensive experimental study, we cannot simply claim that this situation can always be generalized to other complicated applications.

Nevertheless, we still believe that these initial results indicate a promising prospect for our method. Actually, even if the selected MRs are not always very effective, our method has provided a solution for programs without oracle, which is at least better than the inapplicable conventional SBFL.

2) *Internal validity*

The primary threat to the internal validity involves the correctness of our experiment framework which includes the generation of source test cases and the corresponding follow-up test cases with respect to a given MR, generation of mutants, execution of these mutants with both source and follow-up test cases, as well as examination of the outputs of source and follow-up test cases against the corresponding MR. Our experimental setup is quite different from the conventional SBFL using *e_slice*, thus not much existing set up from SIR can be adopted directly. In order to assure the quality of this experimentation, we have conducted a very thorough unit testing at each step of the implementation as well as a comprehensive functional testing at the system level.

3) *Construct validity*

The primary threat to the construct validity is the use of the $\mathcal{EXAM}$ score as a measure of the effectiveness of a SBFL technique. However, this score (or a similar one) has been extensively used in previous studies such as [7, 9, 12, 13, 38], so the threat is acceptably mitigated.

VI. CONCLUSION AND FUTURE WORK

The approach of SBFL has been extensively investigated, and many effective SBFL techniques have been developed. However, all these techniques have assumed that there exists a testing oracle. Since in practice many application domains have the oracle problem, this has severely restricted the applicability of the existing SBFL techniques. Recently, MT has been proposed to alleviate the oracle problem. Thus, it is natural to consider how MT could be used to alleviate the oracle problem in the area of fault localization.

In this paper, we propose a novel concept, *mice*, based on the integration of metamorphic relations and slices. In our proposal, for any existing SBFL techniques, the role of slice is replaced by that of *mice*, the role of an individual test case is replaced by that of a metamorphic test group, and the testing result of *failure* or *pass* of an individual test case is replaced by the metamorphic testing result of *violation* or *non-violation* of an MR. With these three one-to-one replacements, we could then construct metamorphic versions of the existing SBFL techniques.

An experimental study on 9 programs of varying program sizes has been conducted to investigate the performance of our proposed method. From the results, we can conclude that for programs with oracle problem where the conventional SBFL cannot be adopted in practice, our method with *e_mice* is applicable. Moreover, its performance is very similar to the conventional SBFL performance evaluated with an assumed oracle. In other words, our study demonstrates that testing oracles are no longer mandatory in SBFL.

In our future work, we will continue to conduct additional experimental studies on more large-sized programs, more types of faults, as well as the programs with multiple-faults. The effectiveness of different MRs, and how to select better MR to provide a better fault localization performance, will also be studied. Furthermore the proposed method in this paper only addresses one application of *mice*. We believe that there are many interesting applications of *mice* for us to explore.

ACKNOWLEDGMENTS

This project is partially supported by ARC Discovery Project (DP0771733), the National Natural Science Foundation of China (90818027, 60873050), the National High-Tech Research Subjects and Development Plan of China (2009AA01Z147), the National Grand Fundamental Research 973 Program of China (2009CB320703), the Key Laboratory Funding of Science and Technology Commission of Shanghai Municipality (09DZ2272600), the 863 Plan Program of China (2008AA01Z142), and the USA National Science Foundation (NSF DUE-1023071).

REFERENCE

[1] J. S. Collofello and S. N. Woodfield, "Evaluating the effectiveness of reliability-assurance techniques," *The Journal of Systems and Software*, vol. 9, no. 3, pp. 191–195, 1989.

[2] W. E. 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.

[3] 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 (TAICPART-MUTATION)*, Windsor, UK, 2007, pp. 89–98.

[4] 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 (DSN)*, Washington, DC, USA, 2002, pp. 595–604.

[5] 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 (ICSE)*, Florida, USA, 2002, pp. 467–477.

[6] 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 (PRDC)*, Riverside, USA, 2006, pp. 39–46.

[7] 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 (PLDI)*, Chicago, USA, 2005, pp. 15–26.

[8] B. R. Liblit, “Cooperative bug isolation,” Ph.D. dissertation, University of California, 2004.

[9] C. Liu, X. Yan, L. Fei, J. Han, and S. Midkiff, “SOBER: statistical model-based bug localization,” *IEEE Transactions on Software Engineering*, vol. 32, no. 10, pp. 831–848, 2006.

[10] H. Pan and E. Spafford, “Heuristics for automatic localization of software faults,” Technical Report SERC-TR-116-P, Purdue University, Tech. Rep., 1992.

[11] 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.

[12] W. E. 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.

[13] W. E. Wong, T. Wei, Y. Qi, and L. Zhao, “A crosstab-based statistical method for effective fault localization,” in *Proceedings of the 1st International Conference on Software Testing, Verification and Validation (ICST)*, Lillehammer, Norway, 2008, pp. 42–51.

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

[15] J. Baker and J. Thornton, “Software engineering challenges in bioinformatics,” in *Proceedings of the 26th International Conference on Software Engineering (ICSE)*, Scotland, UK, 2004, pp. 12–15.

[16] T. Y. Chen, J. W. K. Ho, H. Liu, and X. Y. Xie, “An innovative approach for testing bioinformatics programs using metamorphic testing,” *BMC bioinformatics*, vol. 10, no. 1, pp. 24–35, 2009.

[17] J. W. K. Ho, M. W. Lin, S. Adelstein, and C. G. dos Remedios, “Customising an antibody leukocyte capture microarray for systemic lupus erythematosus: beyond biomarker discovery,” *Proteomics-Clinical Applications*, vol. in press, 2010.

[18] J. W. K. Ho, M. Stefani, C. G. dos Remedios, and M. A. Charleston, “Differential variability analysis of gene expression and its application to human diseases,” *Bioinformatics*, vol. 24, pp. 390–398, 2008.

[19] E. J. Weyuker, “On testing non-testable programs,” *Computer Journal*, vol. 25, no. 4, pp. 465–470, November 1982.

[20] X. Y. Xie, J. W. K. Ho, C. Murphy, G. Kaiser, B. W. Xu, and T. Y. Chen, “Application of metamorphic testing to supervised classifiers,” in *Proceedings of the 9h International Conference on Quality Software (QSIC)*, Jeju, Korea, 2009, pp. 135–144.

[21] 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.

[22] A. Gotlieb and B. Botella, “Automated Metamorphic Testing,” in *Proceedings of the 27th Annual International Conference on Computer Software and Applications (COMPSAC)*, Dallas, TX, USA, 2003, pp. 34–40.

[23] W. Chan, S. Cheung, and K. Leung, “A metamorphic testing approach for online testing of service-oriented software applications,” *International Journal of Web Services Research*, vol. 4, no. 2, pp. 61–81, 2007.

[24] C. Murphy, K. Shen, and G. Kaiser, “Using JML runtime assertion checking to automate metamorphic testing in applications without test oracles,” in *Proceedings of the 2nd International Conference on Software Testing,Verification and Validation (ICST)*, Denver, Colorado, USA, 2009, pp. 436–445.

[25] ——, “Automatic system testing of programs without test oracles,” in *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA)*, Chicago, IL, USA, 2009, pp. 189–200.

[26] C. Murphy, “Metamorphic testing techniques to detect defects in applications without test oracles,” Ph.D. dissertation, Columbia University, 2010.

[27] T. Y. Chen, S. C. Cheung, and S. M. Yiu., “Metamorphic testing: a new approach for generating next test cases.” Technical Report HKUST-CS98-01, Dept. of Computer Science, Hong Kong Univ. of Science and Technology, Tech. Rep., 1998.

[28] T. Y. Chen, T. H. Tse, and Z. Q. Zhou, “Semi-proving: an integrated method for program proving, testing, and debugging,” *accepted to appear in IEEE Transactions on Software Engineering*.

[29] D. Binkley and M. Harman, “A survey of empirical results on program slicing,” *Advances in Computers*, vol. 62, pp. 105–178, 2004.

[30] D. Binkley, S. Danicic, T. Gyimóthy, M. Harman, Á. Kiss, and B. Korel, “Theoretical foundations of dynamic program slicing,” *Theoretical Computer Science*, vol. 360, no. 1-3, pp. 23–41, 2006.

[31] D. Binkley, N. Gold, and M. Harman, “An empirical study of static program slice size,” *ACM Transactions on Software Engineering and Methodology*, vol. 16, no. 2, p. 8, 2007.

[32] SIR, “http://sir.unl.edu/php/index.php.”

[33] H. Jiang and W. H. Wong, “SeqMap: mapping massive amount of oligonucleotides to the genome,” *Bioinformatics*, vol. 24, pp. 2395–2396, 2008.

[34] J. H. Andrews, L. C. Briand, and Y. Labiche, “Is mutation an appropriate tool for testing experiments?” in *Proceedings of the 27th International Conference on Software Engineering (ICSE)*, St. Louis, MO, USA, 2005, pp. 402–411.

[35] SeqMap, “http://biogibbs.stanford.edu/ jiangh/SeqMap/.”

[36] SLOCCount, “http://www.dwheeler.com/sloccount/.”

[37] GNU, “http://www.gnu.org/software/grep/.”

[38] 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 (ASE)*, Long Beach, California, USA, 2005, pp. 273–282.

[39] J. Jones, J. Bowring, and M. Harrold, “Debugging in Parallel,” in *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA)*, New York, NY, USA, 2007, pp. 16–26.