Technical Report UTDCS-07-10

Spectrum-Based Fault Localization without Test Oracles

Xiaoyuan Xie[1, 3, 4] W. Eric Wong[2] Tsong Yueh Chen[1] Baowen Xu[4]
xxie@groupwise.swin.edu.au ewong@utdallas.edu tychen@groupwise.swin.edu.au bwxu@nju.edu.cn

February 2010

Department of Computer Science
University of Texas at Dallas

[1] Centre for Software Analysis and Testing, Swinburne University of Technology,
Hawthorn, Victoria, 3122, Australia
[2] Department of Computer Science, University of Texas at Dallas,
Richardson, TX 75083, United States
[3] School of Computer Science and Engineering, Southeast University,
Nanjing 210096, China
[4] State Key Laboratory for Novel Software Technology & Department of Computer Science
and Technology, Nanjing University, Nanjing, 210093, China

ABSTRACT

Spectrum-Based Fault Localization (SBFL) is one of the most promising approaches towards fault localization, and has received a lot of attention due to its simplicity and effectiveness. It utilizes various program spectra and the associated testing result of each individual test case, namely *failed* or *passed*, to evaluate the risk of containing a fault for each program entity with different statistical formulas. However, it suffers from a crucial problem which makes it infeasible in many application domains, that is, its assumption of the existence of test oracle. In practice, there are many programs in various domains without such oracles, where obviously, the existing SBFL techniques cannot be applied at all. To address this problem, we introduce a novel type of slices, metamorphic slice (*mslice*), which is property based, as different from the traditional test case based slices in SBFL. We propose to use *mslice* instead of slice, and its associated metamorphic testing (MT) result of a metamorphic test group, which is either *violated* or *non-violated*, rather than the testing result of *failed* or *passed* for individual test cases, to evaluate the risk for each program entity of being faulty. In this way, we can extend SBFL to application domains without test oracles, needless to say that our proposed method is still applicable if the application domains have test oracles. We use a popular UNIX utility program, ***grep***, in our case study to evaluate the effectiveness of the proposed method. With three popular risk formulas (Orchiai, Jaccard and Tarantula), we compare the effectiveness of using *mslice* and traditional slice in SBFL. The empirical results suggest that for test suites of the same size, there is no significant difference between *mslices* and traditional slices.

Keywords: Spectrum-based fault localization, test oracle, metamorphic testing, metamorphic slice, slice

1. 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 [10]. 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 major 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 as SBFL in this paper). Generally speaking, this approach utilizes various program spectra acquired dynamically from software testing, as well as the associated testing result, in terms of *failed* or *passed*, for each test case. The program spectrum can be any granularity of program entities. One of the most widely adopted spectra is the execution slice (denoted as *e_slice*) of each test case, which records those statements executed in one test execution [25].

After collecting this necessary information, SBFL uses different statistical formulas to evaluate the risk of containing a fault for each program entity, and gives a risk ranking list. SBFL intends to highlight program entities which strongly correlate with program failures. These program entities are regarded as the likely faulty locations [2]. Some typical statistical formulas include Pinpoint [4], Tarantula [16], Ochiai [1], and others [17-21, 24, 26, 28].

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

SBFL has received a lot of attention due to its simplicity and effectiveness. However there are still some problems in this approach. And the assumption of the existence of test oracle is one of the most crucial problems, which makes it infeasible in many application domains.

However, there are many real-world programs of which the correctness of their computed outputs are unable or too expensive to be verified. These programs include complex computational programs, machine learning algorithms, etc [3, 6, 13, 14, 23, 27].

In this paper, we propose a new type of slice, metamorphic slice (*mslice*), which is property based and can be used to alleviate the oracle problem in SBFL. Each *mslice* is related to a particular property of the algorithm being implemented. This property is referred to as the metamorphic relation (MR) which involves multiple inputs and their outputs of the algorithm. If the implementation of the algorithm is correct, MR must be satisfied with respect to the relevant multiple inputs and their computed outputs. In other words, for a particular MR, given a particular group of test cases, a testing result about the satisfaction of the MR which is either *violated* or *non-violated*, could then be determined. It should be noted that *violation* or *non-violation* of the MR could be determined even if we do not know whether the testing result is *failed* or *passed* for each element of this group of test cases. Hence, the oracle problem could then be alleviated if the information of *violation* or *non-violation* of the MR is used instead of *failed* or *passed* for an individual test case in SBFL. In other words, we can extend the application of existing SBFL techniques to programs without test oracles.

As the first comparison between SBFL using *mslice* and conventional slice, we focus on the binary coverage information at statement level, which constructs program spectrum using the execution metamorphic slice (denoted as *e_mslice*), which will be formally defined in Section 3.1. In our case study, we investigate a widely used utility ***grep***, which suffers from the oracle problem in parts of its functionalities. For example, it is known that ***grep*** can search the input files for lines containing a match to a given pattern list [11]. However for some command-line options, even if we can manually check whether the listed strings really match the given pattern, we have no way to automatically check whether ***grep*** has output all the strings which match the given pattern, without conducting an exhaustive comparison. With three most popular risk evaluation formulas, the empirical results reveal that for test suites of the same size, SBFL using *mslice* has similar performance as compared with SBFL using conventional slice. As a consequence, even for the case of having test oracles, it is worthwhile to conduct SBFL using *mslice*.

The rest of this paper is organized as follows. Section 2 describes the background of spectrum-based fault localization and metamorphic testing. Section 3 introduces the concept of metamorphic slices, and also describes how metamorphic slices could be used to alleviate the oracle problem in SBFL. In Section 4, we present the experimental setup of our case study. We demonstrate and analyze the empirical results in Section 5. Section 6 gives the conclusion and future work.

2. BACKGROUND

2.1 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 [21]. 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 or basic blocks, etc; while the run-time information could be the binary coverage status, the number of time that the entity has been covered, or the program state before and after executing the program entity, etc. In practice, there are many kinds of combinations [12]. The most widely adopted combination involves statement and its coverage status in one test execution [25], which will be used in our study.

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 related to failure, and hence have higher possibility to contain the faults.

Given a program P with n statements and executed by m test cases, Figure 1 shows the essential information required by SBFL. Vector TS contains the m test cases, matrix M^{TS} represents the program spectrum, and R^{TS} records all the testing results associated with individual test cases. 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.

$$P:\left(s_1 \quad s_2 \quad \dots \quad s_n\right)$$

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

Figure 1. Essential information for traditional SBFL

Utilizing this information, SBFL produces a vector which consists of four indexes for each statement s_j, denoted as $A^j =< a^j{}_{ef}, a^j{}_{ep}, a^j{}_{nf}, a^j{}_{np} >$, where a_{ef} and a_{ep} represent the number of test cases that covered the relevant statement and returns a *failed* and *passed* testing result respectively. While a_{nf} and a_{np} stand for the number of test cases that do not execute the relevant statement, and return a *failed* and *passed* testing result respectively. A risk evaluation formula f is used to map the vector $A^j =< a^j{}_{ef}, a^j{}_{ep}, a^j{}_{nf}, a^j{}_{np} >$ for each statement s_j to a risk value r_j. Existing evaluation formulas include Jaccard [4], Ochiai [1], Ample [28], Tarantula [16], Wong [24], etc. 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.

SBFL is widely adopted because it is simple in concept and easily applied. Furthermore, experimental analysis shows that it is effective. However, there are also some problems in SBFL, and one of the crucial problems is the assumption of the existence of test oracle.

Actually in real-world applications, there are many programs suffering from the oracle problem, that is, it is impossible or too expensive to verify the correctness of the computed outputs. For example, in programs computing multiple precision arithmetic, the operands involved are very large numbers and, hence, the results are very expensive to check. And when testing a compiler, it is not easy to verify whether the generated object code is equivalent to the source code. Besides in object-oriented programs, it is usually very difficult to decide whether two objects are equivalent. Other examples include testing programs involving machine learning algorithms, simulations, combinatorial calculations, graph display in the monitor, etc [3, 6, 13, 14, 23, 27, 29]. In such cases, current SBFL techniques cannot be applied.

2.2 Metamorphic testing

Metamorphic testing (MT) [5, 7] is a testing approach designed 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.

Let us use an example to illustrate MT informally. Readers who are interested in a more formal and comprehensive description of MT, may consult [5, 7]. 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. Given a weighted graph G, a start node x, and a destination node y in G, the target program is to output the shortest path and its length. Let us denote the length of the shortest path by $d(x, y, G)$. 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 problem 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 any node in the shortest path with x as the start node and y as the destination node, then the sum of the length of the shortest path from x to w and the length of the shortest path from w to y shall be equal to the length of the shortest path from x to y, that is, $d(x, y, G) = d(x, w, G) + d(w, y, G)$. 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 $d(w, y, G)$), 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 the previous 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 case but also on the relevant MR. 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.

Generally speaking, when conducting MT, the testers first need to identify an MR of the software under test, and choose a test case selection strategy to generate source test cases. For convenience of reference, we will refer a source test case (or a group of source test cases if appropriate as explained in the previous paragraph, source test cases may be multiple for a specific MR) 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, the correctness of the output of each individual test case needs not be verified. Instead, the MR is verified with respect to the metamorphic test group and its outputs. *Violation* of MR implies an incorrect program.

It should be obvious to see that metamorphic testing is simple in concept, easily automatable, and independent of any particular programming language. Nevertheless, it is not a trivial task to choose an MR which is effective to reveal failure. But, the problem of the choice of MR is beyond the scope of this paper. Interested readers may consult [7-9].

3. APPROACH

In order to make existing SBFL techniques be feasible for the application domains without oracles, we first need to define a new type of "slices", namely, metamorphic slices, in Section 3.1. With the notion of metamorphic slices, we present in Section 3.2 how to extend the application of the SBFL approach beyond the programs that must have test oracles.

3.1 Metamorphic slices

In traditional program slice family, there are three types of slices, namely static slice, dynamic slice, and execution slice [25]. Dynamic slice and execution slice are normally used in SBFL. Their definitions are as follows:

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

Apparently these definitions are based on traditional testing techniques, which involve single test execution and its execution result. However, in metamorphic testing, situations are different. One metamorphic test run involves multiple test executions, and a collective testing result of a metamorphic test group, which is either *violation* or *non-violation* of the MR. We need not to know about the correctness of the output of individual test case of the metamorphic test group. Thus, it is natural to consider how to make use of the notion of MR to alleviate the oracle problem in SBFL.

Thus, we propose the following definitions for a new type of metamorphic slices. Given a metamorphic relation *MR*, a set of *ks* source test cases $T_i^S = \{t_1^S, \ldots, t_{ks}^S\}$, and the corresponding set of *kf* follow-up test cases $T_i^F = \{ t_1^F, \ldots, t_s^F \}$.

- For a given variable *v*, the dynamic metamorphic slice $d\_mslice(v, MR, T_i^S)$ is defined as the set of statements which have actually affected the specified variable *v* in the execution of any test case of the metamorphic test group. Immediately from this definition, we have
  $d\_mslice(v, MR, T_i^S) = (\bigcup_{k=1}^{ks} d\_slice(v, t_k^S)) \cup (\bigcup_{k=1}^{kf} d\_slice(v, t_k^F))$.
- An execution metamorphic slice $e\_mslice(MR, T_i^S)$ is defined as a set of statements which have actually been covered by the current metamorphic test run. Immediately from this definition, we have
  $e\_mslice(MR, T_i^S) = (\bigcup_{k=1}^{ks} e\_slice(t_k^S)) \cup (\bigcup_{k=1}^{kf} e\_slice(t_k^F))$.

Immediately from the above definitions, we have the following relation between *e_mslice* and *d_mslice* that for any variable *v*, $d\_mslice(v, MR, T_i^S) \subseteq e\_mslice(MR, T_i^S)$.

As can be seen from the above definitions, different from the conventional slices which involve single test case, *mslice* involves more than one test case. However, it is not the union of some arbitrarily chosen slices of individual test case, as such an arbitrary union may be intuitively meaningless. Instead, all dynamic slices or execution slices which are grouped together to form the relevant *mslices*, are related to a specific MR. This MR provides a collective test result of *violated* or *non-violated*, which can be treated as the counterparts or alternatives of *failed* or *passed* in existing SBFL techniques. As a reminder, *mslice* is different from the concept of program dice, which also involves multiple slices. The most important difference is: slices forming *mslices* are related to a property of the algorithm being implemented, but slices forming a dice do not have such a binding.

It should be noted that *mslices* are also expected to have various applications as conventional slices. In this paper, we only investigate the applications of *mslices* in SBFL.

3.2 SBFL using *mslice*

Traditionally, in SBFL, two kinds of essential information are required to evaluate the risk of each program statement using various statistical formulas, namely program spectrum and its associated testing results. However, for programs without test oracle, the absence of the information about testing results makes SBFL not applicable. In this Section, we are going to present how the use of *mslice* instead of conventional slice can make SBFL become applicable to the programs without test oracles. As a consequence, the unavailable information about testing result of individual test case can now be compensated by the information about *violation* or *non-violation* of MR with respect to a metamorphic test group. In such a way, we can extend the application domain of SBFL from program with test oracles to programs without test oracles.

As a first study about extending the applicability of SBFL, since the most widely adopted spectrum involve *e_slice* [25], we therefore construct spectra using *e_mslice* in this paper.

Without loss of generality, we assume that an MR has only one source test case and one follow-up test case, that is, T_i^S and T_i^F have one and only one element each. Let us use g_i to denote the corresponding metamorphic test group for T_i^S and T_i^F.

Assume that m T_i^S are generated. Hence, there would be m T_i^F and m metamorphic test groups. The program would be executed with all test cases in each metamorphic test group g_i and a metamorphic testing result of being *violated* or *non-violated* would be given to each g_i. Also constructed is the corresponding execution metamorphic slice. Using all these collected information, we can then construct the program spectrum for program P (assume having n statements) and essential information for SBFL as shown in Figure 2.

$$P:\left(s_1 \quad s_2 \quad \dots \quad s_n\right)$$

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

Figure 2. Essential information for SBFL using *e_mslice*

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

Thus, in extending the SBFL to the application domains without test oracles, *e_slice* is replaced by *e_mslice* of a specific MR, an individual test case t_i is replaced by a metamorphic test group g_i, 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 is then applied to compute the risk vector $A^j =< a^j_{ef}, a^j_{ep}, a^j_{nf}, a^j_{np} >$ and then the risk value for each statement s_j.

Actually, SBFL using *e_mslice* has similar intuition as SBFL using *e_slice*. For SBFL using *e_slice*, 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* does not contain a faulty statement, or does contain a faulty statement. Similarly, for SBFL using *e_mslice*, a *violated* metamorphic test group implies that a faulty statement is included in the corresponding *e_mslice*, while a *non-violated* metamorphic test group does not provide a definite conclusion whether the corresponding *e_mslice* does not contain a faulty statement, or does contain a faulty statement, as explained in detail below.

For a *violated* metamorphic test group g, which has t^S and t^F as the source and follow-up test cases respectively, there are three possible situations for the program P:

- $P(t^S)$ *passed* and $P(t^F)$ *failed*, Hence, the *e_slice(*t^S*)* must contain a faulty statement.
- $P(t^S)$ *failed* and $P(t^F)$ *passed*. Hence, the *e_slice(*t^F*)*must contain a faulty statement.
- $P(t^S)$ *failed* and $P(t^F)$ *failed.* Hence, both the *e_slice(*t^S*)* and *e_slice(*t^F*)* must contain a faulty statement.

However, we do not know which of the above situation occurs. But, we know definitely that the union of *e_slice*(t^S)and *e_slice*(t^F), that is, *e_mslice (MR,* T_i^S*)*, must contain a faulty statement.
On the other hand, for a *non-violated* metamorphic test group there are four possibilities:

- $P(t^S)$ *passed* and $P(t^F)$ *failed*
- $P(t^S)$ *failed* and $P(t^F)$ *passed*
- $P(t^S)$ *failed* and $P(t^F)$ *failed*
- $P(t^S)$ *passed* and $P(t^F)$ *passed*

Similar to the scenario of *passed* test case in SBFL using *e_slice*, as we do not know which of the above situation occurs, hence we cannot draw a definite conclusion that either *e_mslice (MR,* T_i^S*)* does not contain a faulty statement, or does contain a faulty statement.

As a reminder, SBFL using *m_slices* can be applied even when the test oracle does exist.

4. EXPERIMENTAL SETUP

4.1 Testing object

In the case study, we choose ***grep,*** a Unix command-line utility program written in C, as our testing object. According to its manual page, "*the **grep** command searches one or more input files for lines containing a match to a specified pattern; and by default it prints the matching lines.*" [11] The reasons why we choose ***grep*** are as follows:

First, some functionalities of ***grep*** suffer from the test oracle problem, making it an ideal candidate for our proposed method which does not require a test oracle. For example, consider the following command:

```
    Grep –E “[Gg]r?ep” input_file.txt
```

Based on the regular expression"*[Gg]r?ep*", we notice that the command should print out all lines in the file *input_file.txt* which contain "*grep*", "*Grep*", "*gep*", or "*Gep*". However, unless we do an exhaustive examination of the entire file (namely, inspecting every single line of *input_file.txt*), we cannot determine whether really all such lines have been output. As a result, we may only be able to conduct testing on ***grep*** for some special cases whose outputs can be easily verified. This also implies that neither random nor comprehensive testing can be conducted due to the lack of an appropriate test oracle. Consequently, even if we can collect the *e_slice* for each test case, we still do not know whether the corresponding testing result is passed or failed. Hence, SBFL using the traditional execution slices cannot be applied in this case. On the other hand, this is not an issue in our method of using *mslices.*

Second, as compared with the other publicly available programs such as those in the *Siemens suite* which barely have any documentation, ***grep*** has a well-written manual page with a clear description of all its functionalities. This makes it much easier for readers to understand the MRs defined in Section 4.2.

In our study, we used version v0 of ***grep*** 1.2 downloaded from the SIR website [22]. All source files were concatenated into one *grep.c* file; and all executions were on a cluster of 64-bit Intel Clovertown systems running CentOS 5. The statement coverage is collected by using ***gcov***.

4.2 Definition of MR

The ***grep*** program has a large set of functionalities. Instead of examining every one of them to make the experiment unnecessarily complicated with a risk of distracting readers' attention on how *mslice* can be used to help programmers locate software bugs without requiring a test oracle, we only focus on the regular expression analyzer – one of the most important functionalities of ***grep***.

We define three MRs for the regular expression analyzer and impose the following restrictions on the command line:

- The option is fixed as `−E`, that is, using the extended character set in regular expressions, and the output should list all the matching lines in the input file.
- The regular expression is enclosed with double quotes such as `“RE”`.
- The input file is fixed as `“sf.in`

Thus, the command line is like

```
./grep -E “RE” sf.in
```

In this experiment, the used MRs only involve one source test case and one follow-up test case. The metamorphic test group can then be expressed as $g = (t^s, t^f)$. The command line for t^s is

```
./grep -E “REs” sf.in
```

and the output is denoted as O^s. Similarly, the command line for t^f is

```
./grep -E “REf” sf.in
```

And the output is denoted as O^f. All the used MRs are actually different forms of equivalence relationship between source regular expression RE_s and the follow-up regular expression RE_f. Hence, the outputs for these two command lines (namely, O^s and O^f) should be exactly the same. The detailed definitions of each MR are as follows:

- **MR1: Complete decomposition of the brackets structure**

In this MR, the source regular expression RE_s is required to contain a bracket sub-expression such as “[*x*–*y*]”, where *x* and *y* are digits and $x<y$. This regular expression should match a string which has any character within the range of [*x*, *y*]. And the follow-up regular expression RE_f is derived from RE_s by completely decomposing the “[ ]” structure, that is, replacing the character set with its equivalent form using “|”. For example: if RE_s contains “[1–3]”, then in RE_f the corresponding part is substituted with “1|2|3” or “2|1|3”. Note that the order of the digits in the follow-up regular expression is randomly generated. From the specification of ***grep***, we conclude that RE_s is equivalent to RE_f, and consequently the source output O^s should be the same as the follow-up output O^f.

- **MR2: Splitting the square brackets structure**

In this MR, the source regular expression RE_s is required to contain a bracket sub-expression like “[ ]”. The string inside the square brackets can be in any format such as “[1–6]”, “[*abcd*]”, etc. This sub-expression represents a list of characters enclosed by “[”and “]”. It matches any single character in that list. And the follow-up regular expression RE_f is derived from RE_s by splitting the “[ ]” structure at the median position, that is, replacing the character set with its equivalent form by splitting them into two groups using “|”. For example, if RE_s contains “[1–6]”, then RE_f has “[1–3] | [4–6]” in the corresponding part. Or suppose RE_s is “[*abcd*]”, then RE_f can be “[*ab*]||[*cd*]”. Once again, from the specification of ***grep***, we can conclude that RE_s is equivalent to RE_f, and consequently the source output O^s should be the same as the follow-up output O^f.

- **MR3: Bracketing the characters**

In this MR, the source regular expression RE_s is required to only contain the collection of simple character apart from +, *, ?, {}. Besides, RE_s can also start with “^” or “\<”, and end with “$” or “\>”. Some examples of RE_s are “*ab*”, “^*ab**”, or “*abc*{2,3}\>”. The follow-up regular expression RE_f is derived from RE_s by replacing the simple characters with its equivalent form with the added bracket "[ ]". For example, if RE_s contains sub-expression "*abc*", then in RE_f it should be substituted with "[*a*][*b*][*c*]". Obviously, we should have $O^s = O^f$.

Although these three MRs seem to be simple and trivial, they are sufficient to serve the purpose of illustrating our method and its effectiveness.

4.3 Test suite generation

From the descriptions of the three MRs presented in Section 4.2, it is clear that each of them has a very specific requirement on RE_s which implies that the corresponding source test inputs must comply with a certain format. After a careful examination of the 807 test cases posted at the SIR website, which were originally designed for the purpose of code coverage [22], we noticed that very few of them satisfy the required format(s). Another problem with using these tests is that some of them cannot kill any mutants generated with respect to the regular expression analyzer. Hence, in order to have sufficient source test cases for our experiments, we used a test pool with 171,634 random test cases, which was generated and used in another study. Of them, 2,982 test cases are eligible for MR1 (namely, they satisfy the input format requirements imposed by MR1), 5,000 for MR2, and 2,084 for MR3.

4.4 Mutant generation

For version 1.2 of ***grep***, there are 6 versions (v0 to v5) posted at the SIR website [22]. Although some faulty versions (namely, bugs) are also available, very few of them are appropriate for our experiments because they are not related to the regular expression analyzer. More precisely, there are only 10 bugs altogether posted at the SIR website that can be used (4 out of 18 bugs in v1; 0 out of 8 in v2; 4 out of 18 in v3; 2 out of 12 in v4; and 0 out of 1 in v5). For the same reason, most of the bugs described in the Bugzilla database for ***grep*** are not appropriate for our experiments either. As a result, we decided to follow the convention in the SBFL community by using the mutation technique to seed bugs into our testing object, while evaluating the effectiveness of a testing method. Once again, because our MRs are related to the regular expression analyzer, all the bugs seeded by mutation are also in that part.

During the mutant generation, we focus on the first-order mutants (that is, each mutant contains exactly one bug). 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 *logic*) operator.

To generate a mutant, our tool first randomly selects a line in the relevant part within the source code (namely, the regular expression analyzer in this case which spans lines 6605 to 9155 in *grep.c*), 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.

Altogether, 300 mutants were generated. Those which could not be compiled successfully were excluded. Mutants with an exceptional exit that prevented the coverage information from being collected by ***gcov*** were also excluded. Finally, mutants without any *violated* metamorphic test group were excluded as well. As a result, we have 78, 32, and 36 mutants available for MR1, MR2 and MR3, respectively.

4.5 Three risk formulas

In our study, we compare the effectiveness (to be defined below) of using *e_mslice* and *e_slice* in conjunction with three risk formulas listed in Table 1 (namely, Tarantula [16], Jaccard [4] and Ochiai [1]) for software fault localization.

Table 1. Risk evaluation formulas

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

Following other fault localization techniques [15, 17, 19, 24, 26, 28], 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 [15] 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 straightforward to present the percentage of code that *has to be examined* in order to find the fault. This modified score is referred to as the *EXAM* score [24, 26] and is defined (in our case) 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 [24, 26].

5. EXPERIMENTAL RESULTS AND ANALYSIS

5.1 Average Effective Comparison

Experiments with respect to each MR defined in Section 4.2 were conducted for the following four scenarios:

- MS: SBFL using *e_mslice* 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, S-ST and S-FT have the same number of test executions, while the number of test executions of MS and S-AT are double. The numbers of *e_slices* and *e_mslices* used in S-ST, S-FT and MS are the same, while S-AT has used twice as many *e_slices*.

All three formulas listed in Table 1 (Jaccard, Ochiai and Tarantula) were used. In our controlled experiments, we have used the non-mutated version of the ***grep*** program as the test oracle to determine whether the execution of a mutant on a given test case is *passed* or *failed*, that is, for each *e_slice* we know the corresponding testing result (success or failure). However, for MS, such information is not required and is therefore not used at all. Figures 3 to 5 show the average *EXAM* score (i.e., the average percentage of executable statements that have to be examined) for all the mutants with respect to each MR. (Note: each mutant can be viewed as a *bug*.) More precisely, the average score is calculated by dividing the sum of the *EXAM* score of each mutant by the number of mutants.

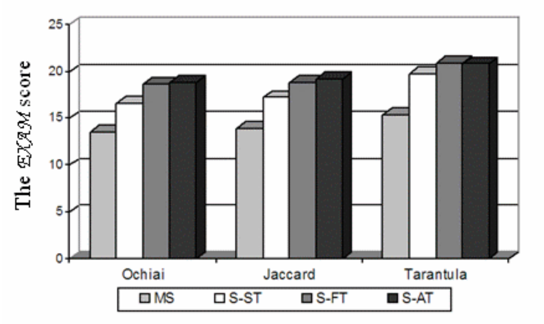

Figure 3. Experiments results for MR1

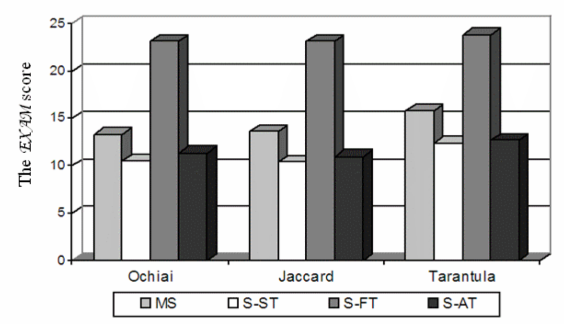

Figure 4. Experiments results for MR2

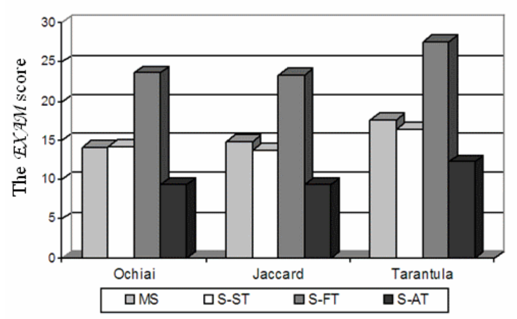

Figure 5. Experiments results for MR3

We make the following observations:

- For MR1, MS is more effective (or better) than S-ST, S-FT, and S-AT (by examining fewer statements) for all formulas.
- For MR2, MS is much more effective than S-FT, but slightly worse than S-ST and S-AT.
- For MR3, MS is much better than S-FT, almost the same as S-ST, but worse than S-AT.

In summary, the effectiveness of MS is comparable with that of S-ST and S-FT (for some cases better than S-FT). Although MS performs worse than S-AT in most cases, it is intuitively expected as more precise information is provided to S-AT than MS.

Table 2 gives the average *EXAM* score for all the mutants used in our study (no matter whether it is for MR1, MR2 or MR3), i.e., the average score calculated by dividing the sum of the average *EXAM* scores in Figures 1, 2 and 3 by 3.

Table 2. The average *EXAM* score for all the mutants

	MS	S-ST	S-FT	S-AT
Ochiai	13.6	18.7	17.9	12.2
Jaccard	14.1	18.6	17.8	12.3
Tarantula	16.3	20.4	20.4	14.8

It is clear that when all the mutants are considered together, MS is more effective than S-ST and S-FT for all three formulas. In each case, the improvement is more than 20%. In particular, this is true for the Ochiai formula. For example, the improvement of MS over S-ST when Ochiai is used is (18.7-13.6)/18.7 = 27.3%. On the other hand, S-AT still performs slightly better than MS for the same reason explained above.

5.2 Detailed Effectiveness Comparison

In this section, we present the detailed effectiveness comparison between MS, S-ST, S-FT and S-AT with respect to each formula listed in Table 1 and all the mutants available for MR1, MR2, and MR3, respectively. For example, Figure 6 gives such a comparison for all the mutants available for MR1 where statement risks are computed using the Ochiai formula. Similar to [24, 26], for a given *x* value, its corresponding *y* value is the cumulative percentage of the mutants (i.e., faults in our case) whose *EXAM* scores are less than or equal to *x*. The curves in this figure are drawn by connecting all the individual data points collected in our study. We observe that MS has 28.2% of the mutants whose *EXAM* scores are less than 5%, and 38.5% of the mutants whose *EXAM* scores are less than 10%. This also implies 10.3% of the mutants whose *EXAM* scores are higher than 5%, but lower than 10%. Similarly, Figures 7 and 8 present the comparisons for mutants available for MR1 and statement risks computed using the Jaccard and Tarantula formulas, respectively.

Figures 6 to 8 are consistent with Figure 3 which show that with respect to the mutants for MR1, MS is the most effective, no matter which of the three formulas is used. For example, MS has almost 30% of the mutants whose *EXAM* scores are less than 5% for all three formulas, whereas S-ST, S-FT and S-AT only have about 10% to 20% of the mutants whose *EXAM* scores are less than 5%.

Figures 9 to 11 present similar comparisons for mutants available for MR2, and Figures 12 to 14 for mutants available for MR3. The observations based on these figures are consistent with those based on Figures 4 and 5. Refer to the discussion in Section 5.1

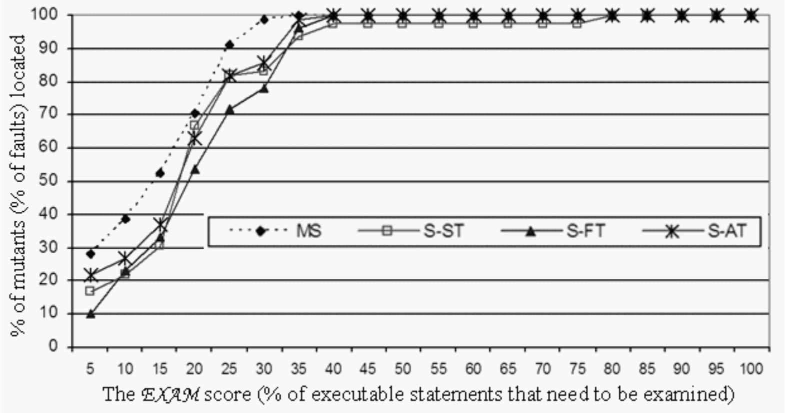

Figure 6. Effective comparison for all mutants in MR1 using Ochiai

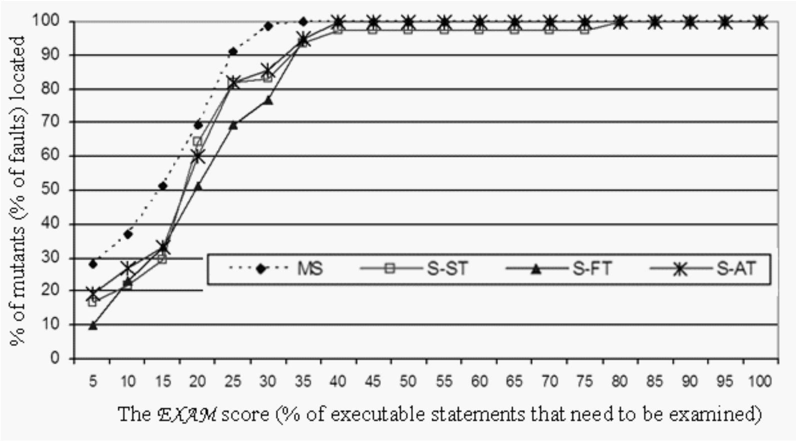

Figure 7. Effective comparison for all mutants in MR1 using Jaccard

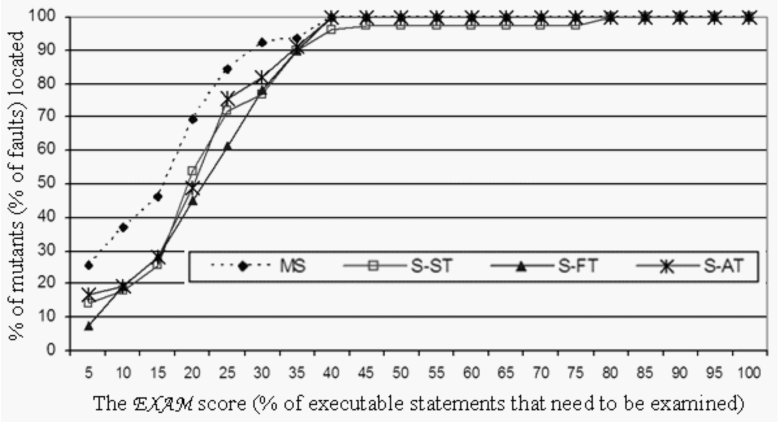

Figure 8. Effective comparison for all mutants in MR1 using Tarantula

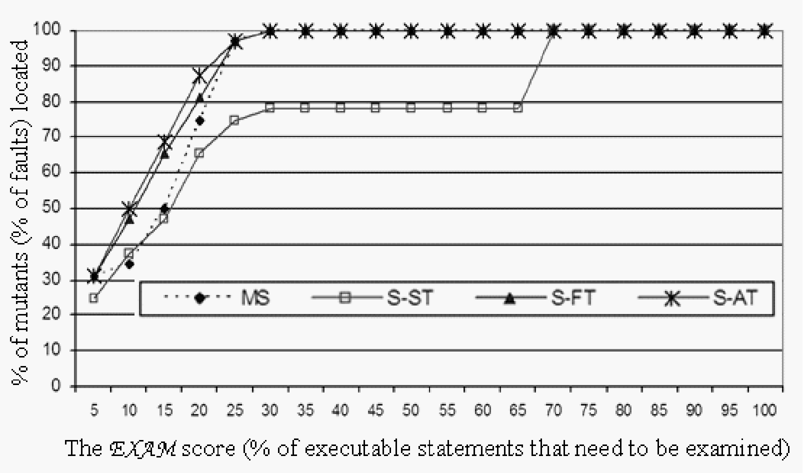

Figure 9. Effective comparison for all mutants in MR2 using Ochiai

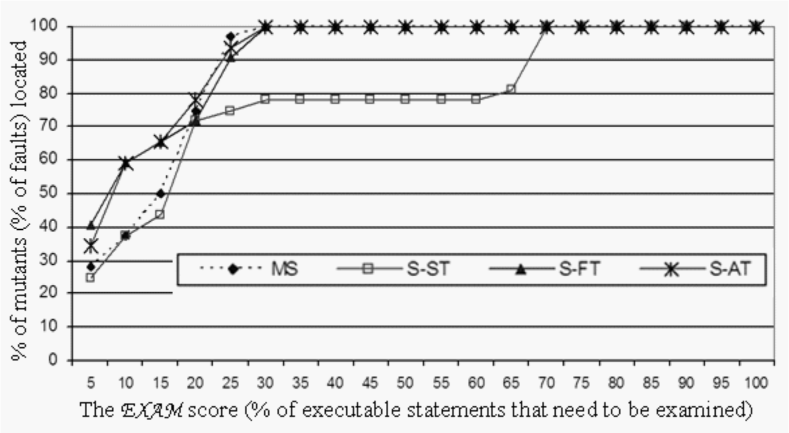

Figure 10. Effective comparison for all mutants in MR2 using Jaccard

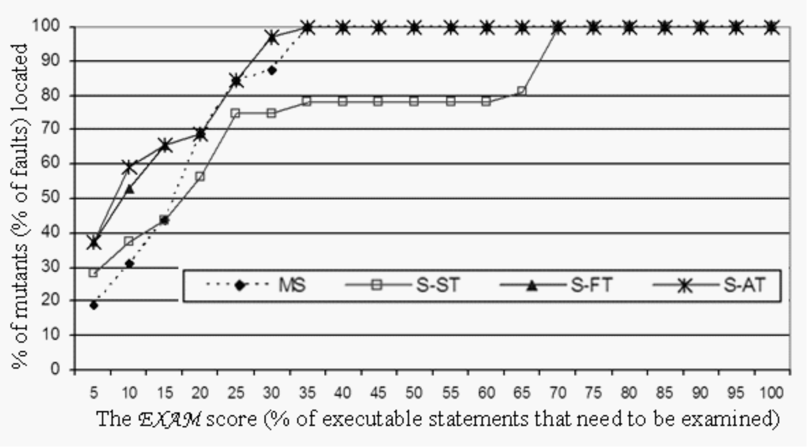

Figure 11. Effective comparison for all mutants in MR2 using Tarantula

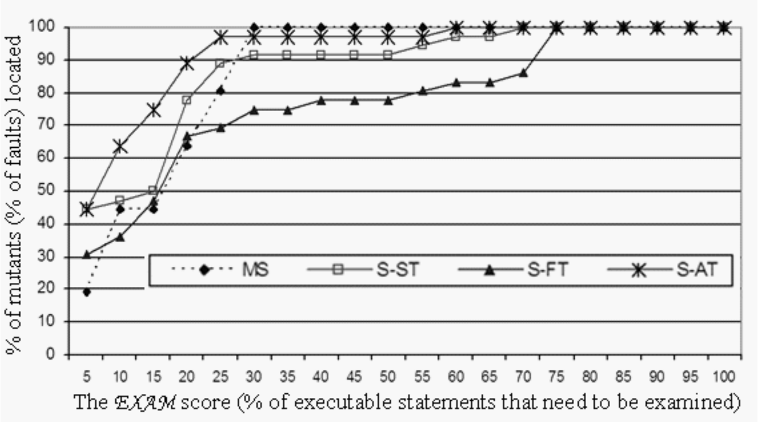

Figure 12. Effective comparison for all mutants in MR3 using Ochiai

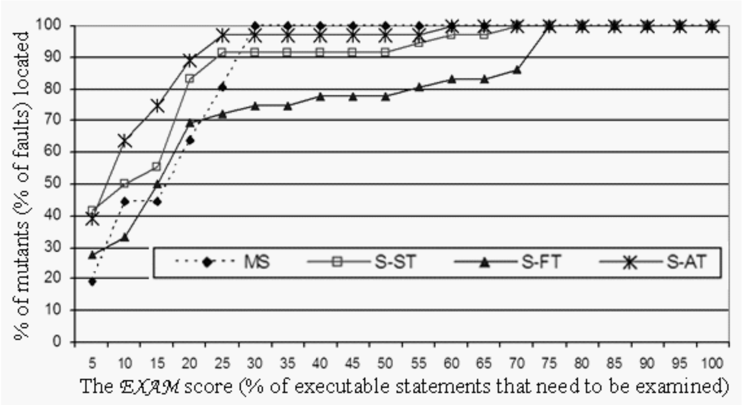

Figure 13. Effective comparison for all mutants in MR3 using Jaccard

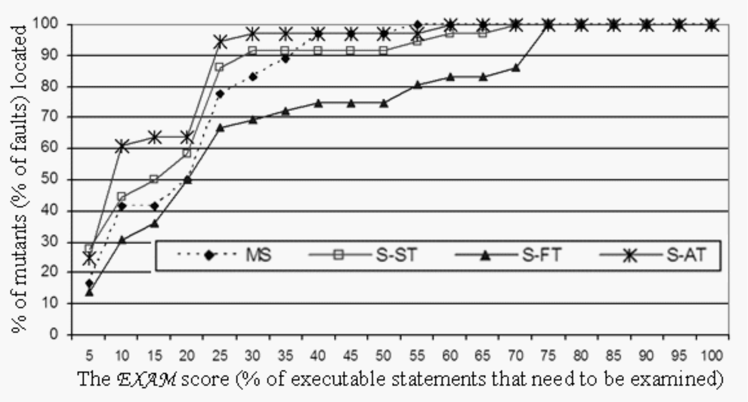

Figure 14. Effective comparison for all mutants in MR3 using Tarantula

5.3 Threats to Validity

5.3.1 External validity

The primary threat to the external validity is the representative of our results acquired from a single testing object, ***grep*** with only three MRs focused on the regular expression analyzer. Although ***grep*** is a real-world program, and is considerably large (in terms of lines of code and the number of executable statements as compared with some publicly available programs such as, the seven programs in the Siemens suite), we still need to use more programs to further validate the effectiveness of our method of using *e_mslice* for locating software bugs. Another threat is the type of mutants (namely, the type of faults) used in our study. Even though these mutants are randomly generated, each mutant, however, only contains exactly one fault and the types of faults are also limited.

Nevertheless, we still believe that our initial results provide a good indication that our proposed technique is useful. In addition, the effectiveness comparison between SBFL using *e_mslice* and SBFL using *e_slice* is also valid. We will leave the complete statistical validation of the results, through the investigation of larger and more complex systems with multiple faults and real bugs in our further study.

5.3.2 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 against 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. This is very different from the SBFL using traditional execution slices. Note that since the faults and test suite posted at the SIR website are not appropriate for our experiments, we had to implement our own tools for test and mutant generation. In order to assure the quality of these tools, we conducted a very thorough unit testing at each step of the implementation as well as a comprehensive functional testing at the system level.

5.3.3 Construct validity

The primary threat to the construct validity is the use of the 𝓔𝓧𝓐𝓜 score as a measure of the effectiveness of a fault localization technique. However, this score (or a similar one) has been used in many studies such as [15, 17, 19, 24, 26], so the threat is acceptably mitigated.

6. CONCLUSIONS AND FUTURE WORK

The approach of SBFL has been extensively investigated, and many effective SBFL techniques were developed. However, all existing SBFL techniques have assumed that there exists a test oracle. Since in practice many application domains have the oracle problems, 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.

With the notion of MT, the concept of *mslice* is developed. The role of *slice* in the existing SBFL techniques can be replaced by that of *mslice,* the role of the testing result of *failed* or *passed* of an individual test case can be replaced by that of the metamorphic testing result of *violated* or *non-violated* of an MR, and the role of an individual test case can be replaced by a metamorphic test group. With these three one-to-one replacements, we could then construct metamorphic versions of the existing SBFL techniques, which are applicable to problem domains that do not have test oracles (as well as the problem domains that have test oracles).

An experimental analysis of a popular Unix utility program ***grep*** has been used to evaluate the effectiveness of our proposed method. The mutation technique is used to seed more bugs in our testing object. The effectiveness of our proposed method is compared with the SBFL using traditional slices, where the currently available version of ***grep*** serves as the test oracle.

From the analysis above, we can conclude that, without the requirement of test oracle, MS can actually achieve quite satisfactory performance. Besides, even to the S-ST and S-FT which rely on the existence of test oracle, the effectiveness of MS is still comparable. On the other hand, though MS performs worse than S-AT in most cases, the significance of MS will not be weakened since S-AT is inapplicable in the applications without test oracles. And the slight decrease in performance of MS is intuitively expected as more *e_slices* information is provided to S-AT than MS.

Overall, SBFL using *mslice* can support a much wider application area, for both the program with and without a test oracle. For programs with test oracles, we can substitute SBFL using traditional slice with *mslice*, which has been demonstrated in this study to have comparable performance. While for programs without test oracles, in which SBFL using traditional slice cannot be applied, we can apply SBFL using *mslice*, to support the fault-localization

The inception of the concept of *mslice* is just another application of MT. In addition to the successful application of MT to programs without test oracles, such as, bioinformatics programs [6], machine learning programs [27], etc., MT has also been used in other testing and verification techniques, such as, fault-based testing techniques [9], symbolic execution [8], etc. Obviously, future work should also include identification of new testing or verification techniques that could be integrated with MT. Furthermore, this paper only addresses one application of *mslice*. We believe *mslice* has other applications similar to its counterpart – traditional slices. Hence, it is interesting to investigate other applications of *mslice***.**

7. ACKNOWLEDGMENTS

This project is partially supported by ARC Discovery Project (DP0771733), as well as 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).

8. REFERENCES

[1] 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)*, pages 39–46, Riverside, USA, 2006. IEEE Computer Society.

[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 (TAICPART-MUTATION)*, pages 89–98, Windsor, UK, 2007. IEEE Computer Society.

[3] J. Baker and J. Thornton. Software Engineering Challenges in Bioinformatics. In *Proceedings of the 26th International Conference on Software Engineering (ICSE)*, pages 12–15, Scotland, UK, 2004. IEEE Computer Society.

[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)*, pages 595–604, Washington, DC, USA, 2002. IEEE Computer Society.

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

[6] 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*, 10(1):24–35, 2009.

[7] T. Y. Chen, F.-C. Kuo, T. H. Tse, and Z. Q. Zhou. Metamorphic Testing and Beyond. In *Proceedings of the 11th Annual International Workshop on Software Technology and Engineering Practice (STEP)*, pages 94–100, Amsterdam, The Netherlands, 2003. IEEE Computer Society.

[8] 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*.

[9] T. Y. Chen, T. H. Tse, and Z. Q. Zhou. Fault-Based Testing Without the Need of Oracles. *Information Software and Technology*, 45(1):1–9, 2003.

[10] J. S. Collofello and S. N. Woodfield. Evaluating the effectiveness of reliability-assurance techniques. *The Journal of Systems and Software*, 9(3):191–195, 1989.

[11] GNU. http://www.gnu.org/software/grep/.

[12] 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*, 10(3):171–194, 2000.

[13] 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*, in press, 2010.

[14] 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*, 24:390–398, 2008.

[15] 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*, pages 273–282, Long Beach, California, USA, November 2005. ACM.

[16] 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)*, pages 467–477, Florida, USA, 2002. ACM.

[17] 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)*, pages 15–26, Chicago, USA, 2005. ACM.

[18] B. R. Liblit. *Cooperative bug isolation*. PhD thesis, University of California, 2004.

[19] C. Liu, X. Yan, L. Fei, J. Han, and S. Midkiff. SOBER: statistical model-based bug localization. *IEEE Transactions on Software Engineering*, 32(10):831–848, 2006.

[20] H. Pan and E. Spafford. Heuristics for automatic localization of software faults. Technical report, Technical Report SERC-TR-116-P, Purdue University, 1992.

[21] 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*, 22(6):432–449, 1997.

[22] SIR. http://sir.unl.edu/php/index.php.

[23] E. J. Weyuker. On testing non-testable programs. *Computer Journal*, 25(4):465–470, November 1982.

[24] 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*, 83(2):188–208, 2010.

[25] W. E. Wong, T. Sugeta, Y. Qi, and J. Maldonado. Smart debugging software architectural design in SDL. *The Journal of Systems and Software*, 76(1):15–28, 2005.

[26] 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),*, pages 42–51, Lillehammer, Norway, April 2008.

[27] 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)*, pages 135–144, Jeju, Korea, 2009. CPS.

[28] A. Zeller. Isolating cause-effect chains from computer programs. *ACM SIGSOFT Software Engineering Notes*, 27(6):10, 2002.

[29] Z. Q. Zhou, D. H. Huang, T. H. Tse, Z. Yang, H. Huang, and T. Y. Chen. Metamorphic Testing and its Applications. In *Proceedings of the 8th International Symposium on Future Software Technology (ISFST)*, pages 346–351, Xi' an, China, 2004. The Software Engineers Association.