2016 IEEE 40th Annual Computer Software and Applications Conference

Code Coverage-based Failure Proximity without Test Oracles

Jingxuan Tu
State Key Laboratory for Novel Software Technology
Department of Computer Science and Technology
Nanjing University
Nanjing, China
Email: tujingxuan@126.com

Xiaoyuan Xie
State Key Lab of Software Engineering
Computer School
Wuhan University
Wuhan, China
Email: xxie@whu.edu.cn

Baowen Xu
State Key Laboratory for Novel Software Technology
Department of Computer Science and Technology
Nanjing University
Nanjing, China
Email: bwxu@nju.edu.cn

***Abstract*—Failure indexing technique plays an important role in modern software maintenance. It can facilitate duplicated failure removal, failure assignment, etc. Failure proximity is a crucial part that underpins failure indexing techniques. It is comprised of two components: a fingerprinting function extracting failure signatures from failures and a distance function computing pairwise distances between failures. Failure proximity usually assumes the existence of test oracle. However, in many real-life application domains, test oracles do not always exist. Hence, the applicability of existing failure proximity techniques is limited. In our paper, we focus on investigating how to apply metamorphic testing on code coverage-based failure proximity without test oracles. In our approach, instead of using the testing results of failure, the results of violation or non-violation for metamorphic test groups are used. Specifically, the fingerprinting function extracts signatures from metamorphic slices rather than execution slices and the distance function computes the pairwise distance between violations rather than between failures. Thereby, the applicability of failure proximity is extended to the situations without test oracles. The experimental results on 50 two-fault mutants show that the quality of proximity matrix obtained through our approach is statistical comparable to traditional code coverage-based failure proximity with test oracle.**

***Keywords*-failure proximity; metamorphic testing; test oracle; metamorphic slice**

I. Introduction

Failure reporting plays an important role in modern software and is widely deployed in software like Mozilla. In practice, end-users are much likely to run into failures of the released software. Under the end-user's permission, failure reporting system automatically collects the information related the failures in a standard form like Bug Report and sends it back to developers for diagnosis. The failure related information like the stack traces, execution traces, etc., is collected every day and the size can be very huge. Hence, in-depth analysis of the failure related information is utilized to aid developers in classifying and diagnosing bugs [1]. Failure indexing technique is commonly utilized to extract failure signatures so as to partition failures due to the same fault [2]. Failure indexing is of key importance in facilitating duplicated failures removal, failure assignment and parallel debugging, etc. [3]. Generally speaking, a failure indexing technique includes three steps: first, using a fingerprinting function to extract failure signatures from related information of program failures like execution traces; second, computes the pairwise distances between failures based on the extracted failure signatures; third, partition the failures based on the distance values matrix (called proximity matrix), then a clustering algorithm can be applied based on the obtained proximity matrix.

The critical part of underpinning an effective failure indexing technique is a proper fingerprinting function and distance computation. Different combinations of fingerprinting function and distance computation comprise different failure proximities and different clustering algorithms can be applied on the same proximity matrix. The central question of failure indexing is to properly design failure proximity which quantifies the probability of two failures being indexed together due to the same bug. A good failure proximity is a proximity matrix that makes the distance between failures due to the same fault small and the distance between other failures large.

In the research work of Liu et al. [2], they investigate an array of six failure proximities which range from the simplest to most complex approaches. Failure point-based failure proximity and stack trace-based failure proximity are two simple approaches for partitioning crashing failures due to the same bug. The fingerprinting functions of the two proximities extract failure points and failure stack traces from each failure respectively and they both adopt the 0-1 distance as the distance function to compute the proximity between any two failures. Code coverage-based failure proximity, predicate evaluation-based failure proximity, dynamic slicing-based failure proximity and statistical debugging-based failure proximity aim to identify noncrashing failures due to the same bug. Code coverage-based failure proximity is simple and widely adopted to measure the proximity between any two failed executions [4–6]. In this paper, our study focuses on code coverage-based failure

0730-3157/16 $31.00 © 2016 IEEE
DOI 10.1109/COMPSAC.2016.81

proximity. The fingerprinting function of code coverage-based failure proximity extracts execution traces from failed test executions and the Jaccard Distance is adopted as the distance function to compute the proximity between any two failures. The detailed description of code coverage-based failure proximity is introduced in Section II.

Generally speaking, code coverage-based failure proximity utilizes fingerprinting function to extract failure signatures from failed test executions. Given any program containing multiple bugs under testing, the associated test results in terms of passing or failing for individual test executions are recorded, correspondingly, failures can also be identified. The previous work of Liu et al. [2, 7] relies on the availability of a test oracle. Only with a test oracle can the failure proximity be computed. However, in many application domains, a test oracle is likely to be too expensive to be obtained or even does not exist, such as machine learning applications [8], bioinformatics programs [9], service-oriented applications [10], etc. In such conditions, failures cannot be identified, correspondingly, the failures due to the same bug cannot be partitioned either. As a consequence, the computation of failure proximity becomes infeasible in these programs and failure indexing techniques cannot be applied either.

Metamorphic testing has been presented to be a simple and effective approach to alleviate the test oracle problem [11]. In many application domains, such as machine learning classifiers [8], bioinformatics programs [9], embedded software [12] etc, metamorphic testing has been applied to alleviate the oracle problem and can effectively detect the bugs. Also, metamorphic testing can be integrated with other software testing and analysis techniques [13–15], so as to extend their applicability to these programs without test oracles.

In this paper, we extend the applicability of code coverage-based failure proximity to the conditions without test oracles. Consequently, we focus on the following two research questions.

1. **RQ1:** How can we extend the applicability of code coverage-based failure proximity to the application domains without test oracles?
2. **RQ2:** How good is the quality of the obtained code coverage-base failure proximity?

The remainder of this paper is organized as follows: Section II introduces the background of code coverage-based failure proximity and metamorphic slice. Section III presents our approaches. In Section IV, we give the set-up for our experimental study, and analyze the experimental results and threats to validity in Section V. The related work is presented in Section VI. Section VII gives the conclusions and future work.

II. Background

A. Code coverage-based failure proximity

Failure proximity is the crucial part of failure indexing techniques and is comprised of two components: fingerprinting function and distance function. Different fingerprinting functions extract different failure signatures which can be expressed in different forms. Different failure signatures can be expressed in different forms like sets, etc. Usually, once the fingerprinting function is determined, the distance function is mostly determined, such as Jaccard distance for sets. Code coverage-based failure proximity extracts coverage information from failed test executions. Notice that coverage information can be obtained at statement level, function level, etc. The code coverage in this study is at statement level. The extracted failure signature of code coverage-based failure proximity is represented in the form of sets. Correspondingly, the Jaccard distance which is the choice for sets is adopted as the distance function in code coverage-based failure proximity. We formally introduce code coverage and Jaccard distance as follows.

Given a program with n statements $PG = \{s_1, s_2, \ldots, s_n\}$, a test suite with l test cases $TS = \{t_1, t_2, \ldots, t_l\}$, for each execution of test cases t_i, a set of executed statements es_i can be collected where $es_i \subseteq PG$. Notice that es_i is also called execution slice. The code coverage information of the program PG under test suite TS can be collected which is the set of all execution slices.

Jaccard distance is proposed by Levandowsky et al. [16] and is defined on sets which is a metric to measure the degree of distinction (also called dissimilarity) between two sets. The following is the formal definition.

Definition 1 (Jaccard distance)**.** *Given any two non-empty sets S_1 and S_2, the Jaccard distance between S_1 and S_2 is computed as: $Distance(S_1, S_2) = 1 - |S_1 \bigcap S_2|/|S_1 \bigcup S_2|$ where $|S|$ denotes the size of set S.*

Let TS_f with the size of m denote the set of failed test cases where $TS_f \subseteq TS$ and C_f denote the set of failed execution slices. Given any two execution slices es_i and es_j belonging to C_f, each execution slice can be regarded a set of statements involved by the corresponding failed test case. Then, the corresponding distance between es_i and es_j can be computed using Jaccard distance as follows.

$$Distance(es_i, es_j) = 1 - \mid es_i \bigcap es_j \mid / \mid es_i \bigcup es_j \mid$$

Finally, a $m \times m$ proximity matrix M is computed, where $M_{i,j}$ is the Jaccard distance between failing execution slices es_i and es_j. With the existence of test oracle, code coverage-based failure proximity can be obtained successfully from the above. However, there exist many software application domains without test oracles, in order to extend the applicability of such failure proximity, we utilize metamorphic testing to achieve this goal introduced in Section III.

B. Metamorphic slice

In order to extend the applicability of SBFL to application domains having test oracle problem, Xie et al. [14] integrated the concept of slices (like execution slices, dynamic slices, etc.) with metamorphic testing and proposed the concept of metamorphic slice.

Firstly, we introduce the basic concept of metamorphic testing. Metamorphic testing (MT) is a methodology for alleviating the test oracle problem [11, 17, 18]. MT generates new test cases from the existing test cases based on the specific properties namely metamorphic relations (MRs), which is extracted from the problem domain. In MT, rather than verifying the correctness of each test execution outputs, MT verifies whether MRs are satisfied between multiple inputs and their expected outputs. The basic process for implementing metamorphic testing is summarized as follows:

(1) Identification of metamorphic relations. Extract necessary properties from the program and take them as the metamorphic relations between multiple inputs and their expected outputs.
(2) Generation of metamorphic test groups. Obtain a set of source test cases for the program under test and construct related follow-up test cases according to the identified metamorphic relations. Each pair of source test case and related follow-up test case is grouped as a metamorphic test group.
(3) Execution of metamorphic test groups. The test outputs of metamorphic test groups are checked whether violate the relevant metamorphic relations where any violation indicates bugs existing in the program.

Secondly, we introduce the concept of metamorphic slice. Generally speaking, a metamorphic slice integrates slices within one metamorphic group that binds source and follow-up test cases according to a specific metamorphic relation.

Formally, given a metamorphic relation denoted as MR1, let $T^S = \{t_1^S, t_2^S, \ldots, t_k^S\}$ be the source test cases, correspondingly, the related follow-up test cases can be constructed by MR1 from T^S, which is denoted as $T^F = \{t_1^F, t_2^F, \ldots, t_n^F\}$. T^S and T^F compose a metamorphic test group (denoted as g) of MR1.

Corresponding to execution slice, execution metamorphic slice is defined. Let es_i^S and es_i^F denote the execution slices of the source test case t_i^S and follow-up test case t_i^F, respectively. Thus, the execution metamorphic slice of g is the union of all execution slices of source and follow-up test cases in this metamorphic test group. The execution metamorphic slice (denoted as $mslice\_e(MR, g)$) is formally defined as follows.

$$mslice\_e(MR, g) = (\bigcup_{i=1}^{k} es_i^S) \bigcup (\bigcup_{i=1}^{n} es_i^F)$$

In [14], Xie et al. showed that metamorphic slice was successfully applied on SBFL techniques where test oracles do not exist. In this paper, we investigate how to apply metamorphic slice on code coverage-based failure proximity where test oracles do not exist.

III. Code coverage-based failure proximity without test oracles

In this section, we describe how to utilize metamorphic testing to alleviate the test oracle problem in code coverage-based failure proximity.

Code coverage-based failure proximity is comprised of two components, namely code coverage fingerprinting function and Jaccard distance function. The proximity matrix quantifies the possibility of any two failures due to the same bug.

For programs without oracle, there are two problems to be addressed in failure proximity: the first problem is how to collect the failures. In traditional code coverage-based failure proximity, failures can be easily collected through comparing the test execution outputs with the expected outputs, however, in the condition of code coverage-based failure proximity without test oracles, the expected outputs are not available. As a consequence, we utilize metamorphic test groups to replace each test case in traditional code coverage-based failure proximity. The metamorphic test results of *violation* or *non-violation* are always available for metamorphic test groups where the violation of a metamorphic test group indicates a failure of the program. After the executions of metamorphic test groups, the metamorphic test groups being violated are collected. The second problem is how to compute the proximity matrix from the violated metamorphic test groups. An execution metamorphic slice binds all execution slices of source and follow-up test cases belonging to a metamorphic test group of MR. Each metamorphic slice is associated with a metamorphic testing result of *violation* or *non-violation*. Thus, for the code coverage-based failure proximity without test oracles, code coverage fingerprinting function extracts execution metamorphic slices from metamorphic test groups and execution metamorphic slices are extracted as the failure signature. The distance between any two violated metamorphic slices is computed using Jaccard distance function. In this way, the proximity matrix is obtained.

Given a program P, a set of source test cases $TS = \{T_1^S, T_2^S, \ldots, T_n^S\}$, the general steps of applying metamorphic slice on code coverage-based failure proximity is shown in Algorithm 1.

It can be found from Algorithm 1 that metamorphic slice is the key component in addressing code coverage-based failure proximity without test oracles. Comparing the traditional code coverage-based failure proximity, we replace the individual test cases, the test results of failed or passed and the failure signature of execution slices by the metamorphic test groups, the metamorphic test results

Algorithm 1 Proximity matrix computation algorithm

Input: Program P, source test suite TS,
Output: Proximity matrix M

(1) Identify metamorphic relations MRs from program P.
(2) Construct related follow-up test cases $TF = \{T_1^F, T_2^F, \ldots, T_n^F\}$ according to MRs from source test cases TS and obtain a set of metamorphic test groups denoted as G.
$G = \{g_1, g_2, \ldots, g_n\}$ where $g_i = (T_i^S, T_i^F)$
(3) Execute program P under G and collect the set of violated metamorphic test groups G_f where $G_f \subseteq G$ and corresponding execution slices of T_i^S and T_i^F where $(T_i^S, T_i^F) \in G_f$.
(4) Compute the execution metamorphic slice for any metamorphic test group $g_i \in G_f$ of each MR. The size of T_i^S and T_i^F in g_i is denoted as k and n respectively.
$mslice\_e(MR, g_i) = (\bigcup_{j=1}^{k} es_j^S) \bigcup (\bigcup_{j=1}^{n} es_j^F)$
(5) Compute the distance between any two violated metamorphic test groups using Jaccard distance.
$Distance(g_i, g_j) = 1 - \frac{|\ mslice\_e_i \bigcap mslice\_e_j\ |}{|\ mslice\_e_i \bigcup mslice\_e_j\ |}$
(6) Return the failure proximity matrix M.

of *violation* or *non-violation* and metamorphic slices respectively. Correspondingly, the computation of distance between any two failures is replaced by the computation of distance between any two violated metamorphic test groups.

Intuitively speaking, **RQ1** has been addressed. But it is necessary to investigate how well our method performs as compared to traditional method. Hence, we conduct experimental studies to further address **RQ2**.

IV. Experimental set-up

A. Experimental objective

In order to investigate the effectiveness of the application of execution metamorphic slice on code coverage-based failure proximity, we choose *grep* as the experimental subject program which has been used in previous MT applications [14, 15] and failure proximity studies [2, 7]. *grep* is a well-known utility program with 7309 lines of code written in C performing pattern matching in Unix. The source code used in our experiment is *grep 2.0*. The subject program can be obtained from Software-artifact Infrastructure Repository (SIR) [19].

The inputs of program *grep* include a pattern to be matched and files for searching. The outputs of program *grep* are all lines containing a match to the input pattern. Given an input pattern to be matched, if the input files for searching is very large, it is actually difficult to check whether the corresponding outputs are the expected ones, because it is almost impossible to check whether all the printed lines by *grep* contain a matched pattern, unless conduct a entire examination of the input files. It indicates that there exists test oracle problem in *grep*.

B. Metamorphic relations definition

The metamorphic relations can be identified from the algorithm of *grep*. We refer interested readers about the detailed specification of *grep* to GNU website [1]. Three MRs are defined which were used in previous work [14]. All the three MRs are equivalent relations, that is given the equivalent input between source test case and related follow-up test case, the output of the follow-up test case is the same as that of source test case.

Let P_s and P_f denote the inputs of source test case and follow-up test case, respectively. Correspondingly, the outputs of P_s and P_f are denoted as O_s and O_f, respectively. Notice that in P_s and P_f the input files for searching is the same and we only transform the input pattern in P_s. The input patterns in P_s and P_f are denoted as p_s and p_f, respectively. The three MRs are described as follows.

MR1: **Decomposition of bracketed sub-expression**.
Given a bracketed sub-expression "$[s_1 s_2 \ldots s_n]$" (denoted as p_s) which is as the source test case, where s_i in p_s is a single character. If the values of these characters in the bracket are continuous, then p_s can be also expressed as "[s_1-s_n]". Notice that orders of characters are according to the ASCII codes. It is not difficult to find out that the equivalent pattern of p_s is the complete decomposition of the bracket by using the operation "|" meaning "or". Thus, MR1 is defined as: given a source test case P_s and the related follow-up test case P_f which is constructed through completely decomposing such bracketed sub-expression in p_s, then the outputs of P_s and P_f should be equal, namely O_s=O_f. Taking for example p_s containing a bracketed sub-expression "[$abcd$]" or "[a-d]", then this sub-expression is replaced by "$a|b|c|d$" in the related P_f.

MR2: **Separation of bracketed sub-expression**
Another equivalent pattern of the sub-expression "$[s_1 s_2 \ldots s_n]$" or "[s_1-s_n]" is splitting the bracket structure into two brackets by using "|". Thus, MR2 is defined as: given a source test case P_s and the related P_f which is constructed through splitting the brackets structure in p_s, then the outputs of P_s and P_f should be equal, namely O_s=O_f. Taking for example p_s containing a bracketed sub-expression "[$abcd$]" or "[a-d]", then this sub-expression is replaced by "[ab]|[cd]" in the related P_f.

MR3: **Brackets addition of simple characters**
Given a simple character which is not a reserved word, the equivalent pattern is enclosing the simple character with brackets. Thus, MR3 is defined as: given a source test case P_s and the related P_f which is constructed through bracketing some simple characters in p_s, then the outputs of P_s and P_f should

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

be equal, namely O_s=O_f. Taking for example a p_s containing a sub-expression "abcd", then this sub-expression is replaced by "[a][b][c][d]" in the related P_f.

C. Evaluation metrics

In this study, we reuse the two metrics proposed in [2, 7] to evaluate the quality of code coverage-based failure proximity without test oracles. The two metrics are proposed to measure the quality of failure proximity: Silhouette Coefficient quantitatively evaluates the goodness of failure proximity; the proximity graph qualitatively compares the failure proximity through visualization.

1) Silhouette Coefficient: Given a program containing k faults, a set of violated metamorphic test groups with the size of m, then $m \times m$ proximity matrix M can be computed from Algorithm 1 in Section III. Let G_l denote the set of violated metamorphic test groups due to the lth fault and $\Phi(g_i)$ denote the index of fault accounting for the violated metamorphic test group g_i. The Silhouette Coefficient for proximity matrix M is the average Silhouette Coefficients of all the violated metamorphic test groups. The Silhouette Coefficient for each violated metamorphic test group is defined as follows.

$$SC(g_i) = \frac{d_i - a_i}{max\{d_i, a_i\}}$$

where

$$a_i = \frac{\Sigma_{g_j \in G_{\Phi(g_i)}} M(i,j)}{|G_{\Phi(g_i)}|}, d_i = \min_{l=1,\ldots,k, l \neq \Phi(g_i)} \frac{\Sigma_{g_j \in G_l} M(i,j)}{|G_l|}$$

Intuitively, in optimal condition, a failure proximity should assign zero distance between violated metamorphic test groups due to the same fault. Silhouette Coefficient evaluates the goodness of a failure proximity through qualifying how far this proximity deviates from the optimal proximity. In the computation of $SC(g_i)$, a_i computes the average distance between g_i and other violated metamorphic test groups due to the same fault and d_i obtains the minimum average distance between g_i and other sets of violated metamorphic test groups due to any different faults. The range of $SC(g_i)$ is from -1 to 1. A positive value closer to 1 represents that g_i is closer to other violated metamorphic test groups due to the same fault than those due to other different faults and vice versa for a non-positive value. Finally, the Silhouette Coefficient for M is computed as follows.

$$SC(M) = \frac{\Sigma_{i=1}^{m} SC(g_i)}{m}$$

It is easy to find out that the value of $SC(M)$ ranges from -1 to 1. A higher Silhouette Coefficient of M indicates a better quality of proximity matrix.

2) Proximity graph: Proximity matrix encodes the proximity between any two violated metamorphic test groups. However, the proximity matrix cannot be visualized directly in the original high dimensional space. MDS techniques [20] are utilized to reduce the proximity matrix to a lower dimensional space (usually two or three), meanwhile, pairwise distances between any two violated metamorphic test groups in M are best preserved. Thus, the reduced dimensional space can be visualized in the form of a proximity graph.

In a proximity graph, each point represents a violated metamorphic test group. The distance between any two points in proximity graph approximately represents the pairwise distance between the two associated violated metamorphic test groups in the proximity matrix. The points associated with the violated metamorphic test groups due to the same fault are shown in the same shape and otherwise in different shapes, hence, the quality of failure proximity could be qualitatively expressed through the corresponding proximity graph visually.

D. Two-fault mutants generation

Since the number of mutants provided by SIR is too small for a mutation analysis, we generate our own faults. Moreover, in our experimental study, we generate two faults in each mutant. In order to obtain two-fault mutants, we first randomly generated a set of single-fault mutants, then randomly combined two single-fault mutants into a two-fault mutant, that is the individual faults of two single-fault mutants are randomly selected to inject into a two-fault mutant. The generation of single-fault mutants focused on non-omission faults and used two types of mutation operators, namely statement mutation and operator mutation. Statement mutation focuses on three types of statements: continue, goto and break statements; operator mutation replaces an arithmetic or logical operator by another arithmetic or logical operator. Thus, we randomly generate 45 single-fault mutants, and based on these single-fault mutants, 255 two-fault mutants are randomly generated.

Among the 255 two-fault mutants, the following types of mutants are excluded: the mutants which could not be compiled successfully or have an exceptional exit; the mutants with no violation of any metamorphic test group; the mutants which could not produce enough eligible violated metamorphic test groups which is introduced in the following part. In summary, for *grep*, we have 19 two-fault mutants for MR1, 15 for MR2 and 16 for MR3. As a reminder, the number of generated two-fault mutants is sufficient for our experimental evaluation.

E. Metamorphic test execution

Before conducting our experimental study, we first obtain the source test cases. In our experiment, the 807 test cases from SIR are not utilized as the source test cases for *grep*,

since the number of source test cases which can be effectively applied our MRs are quite small. Therefore, in order to obtain sufficient source test cases, we used a test pool with 171634 random test cases from the previous studies [14]. Finally, 2982 test cases were obtained as source test cases for MR1, 5003 for MR2 and 2084 for MR3. Accordingly, the related follow-up test cases can be constructed from the source test cases for each of the above three MRs. Thus, there are 2982 metamorphic test groups for MR1, 5003 for MR2 and 2084 for MR3.

Referring to Step 3 in Algorithm 1, after executing all the metamorphic test groups for each mutant, the corresponding violated metamorphic test groups for each mutant are collected. Then the execution metamorphic slice for each violated metamorphic test group can be computed. As a reminder, for clarity and simplicity, similar to the previous work[2], in this study, we only focus on violated metamorphic test groups that are each caused by one and only one fault. Thus, we do not investigate failure proximity for faults correlated semantically without test oracles. Consequently, execution metamorphic slices for the violated metamorphic test groups which are due to two faults are not computed in our study.

Formally, given a two-fault mutant mu which is generated by two single-fault mutants mu_1 and mu_2, let G, G_1 and G_2 denote the violated metamorphic test groups for mu, mu_1 and mu_2 respectively. In our study, we did not compute execution metamorphic slice for each violated metamorphic test group belonging to G. The metamorphic test groups which were prepared to investigate code coverage-based failure proximity without test oracles for mu should satisfy the following conditions: first, the violated test group G should be sufficient to provide enough number of violated metamorphic test groups for each single-fault mutants mu_1 and mu_2; second, the violated metamorphic test groups which are caused by the two faults respectively do not overlap as much as possible so as to minimize the semantic correlation between the two faults. Assume the prepared violated metamorphic test groups of mu, mu_1 and mu_2 to be G', G'_1 and G'_2 respectively where $G' \subseteq G$, $G'_1 \subseteq G_1$ and $G'_2 \subseteq G_2$, then we have $G' = G'_1 \bigcup G'_2$ and $G'_1 \bigcap G'_2 = \emptyset$. The violated metamorphic test groups in G which do not satisfy the above conditions are excluded. Notice that any two-fault mutant which has few satisfied violated metamorphic test groups is excluded in the two-fault mutants generation naturally.

V. Empirical results

To extend the applicability of code coverage-based failure proximity without test oracles, metamorphic slice is applied on failure proximity. In this section, we aim to investigate the quality of the failure proximity obtained through our approach when test oracles do not exist. The proximity matrix can be evaluated quantitatively and qualitatively through Silhouette Coefficient and proximity graph, respectively. The results of the two metrics in traditional failure proximity with test oracle are applied as a benchmark. Thus, we investigate whether code coverage-based failure proximity with metamorphic slice have similar quality to the traditional one. If there is no significant difference in Silhouette Coefficient, then the proximity graph should be similar, then metamorphic slice can be considered to be a successful application on code coverage-based failure proximity without test oracles. As a reminder, the investigation is conducted based on the prepared violated metamorphic test groups which are introduced in Section IV. In summary, the failure proximity is computed under the four following scenarios.

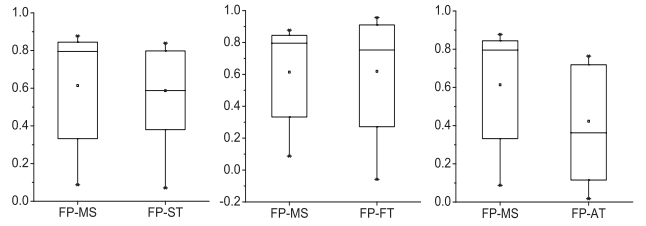
(1) FP-MS: failure proximity using metamorphic slices with all prepared violated metamorphic test groups.
(2) FP-ST: failure proximity using execution slices with all source test cases in the prepared violated metamorphic test groups.
(3) FP-FT: failure proximity using execution slices with all follow-up test cases in the prepared violated metamorphic test groups.
(4) FP-AT: failure proximity using execution slices with all source and follow-up test cases in the prepared violated metamorphic test groups.

In our experiment, scenario FP-MS is adopted as a experimental group where the test oracle does not exist, while scenarios FP-ST, FP-FT and FP-AT are adopted as control groups where the test oracle is required. In the experimental group, we applied our approach and the test oracle is imitated through the violation or non-violation of metamorphic test groups; in the three control groups, the test execution results of non-mutated version are adopted as test oracles. Notice that the amount of prepared violated metamorphic test groups in FP-MS, FP-ST and FP-FT is the same, thus the amount of raw data for the failure proximity is the same and their proximity quality evaluation is expected to have the same degree of reliability; the number of test executions in FP-MS and FP-AT are the same and thus the two scenarios have the same program execution overheads.

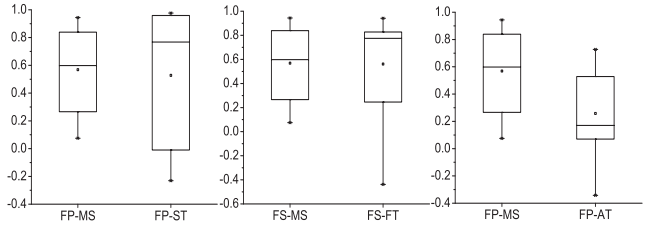
A. Silhouette Coefficient

In order to investigate whether code coverage-based failure proximity without test oracles has similar proximity quality to the traditional one, Silhouette Coefficients between FP-MS and FP-ST, FP-MS and FP-FT, FP-MS and FP-AT are statistically compared respectively in this part. The overall results are shown in the form of boxplot, then the paired Wilcoxon-Signed-Rank test is utilized to conduct the statistical comparisons.

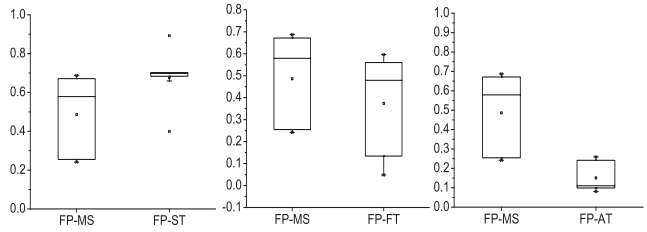
Boxplot provides basic information for data distribution. The overall results are shown in Fig. 1. For each MR, the data distribution of Silhouette Coefficients between FP-MS and the other three scenarios are shown in the form of



(a) MR1



(b) MR2



(c) MR3

Figure 1. Overall Results for MR1, MR2 and MR3

boxplots. From Fig. 1, we observe that for MR1 and MR2, FP-MS has similar data distribution as that of FP-ST and FP-FT. It indicates that FP-MS is likely to have similar proximity matrix quality as FP-ST, FP-FT for MR1 and MR2. For MR1 and MR2, the first quartile, median and third quartile of FP-MS are all above those of FP-AT; for MR3, the three values of FP-MS are above those of both FP-FT and FP-AT. Thus, FP-MS is likely to have better proximity quality than that of FP-AT for MR1, MR2 and MR3, and that of FP-FT for MR3. For MR3, the first quartile, median and third quartile of FP-MS are all below those of FP-ST, thus, FP-MS is likely to have worse proximity quality than that of FP-ST for MR3. From Fig. 1, we can have a direct visual comparison. However, the results of Fig. 1 are not rigorous enough. Furthermore, we apply the paired Wilcoxon-Signed-Rank test to conduct statistical comparisons which are more scientific.

The paired Wilcoxon-Signed-Rank test is a non-parametric hypothesis testing for the differences between two groups of data $F(x)$ and $G(y)$ which do not follows a normal distribution [21]. To obtain a conclusion, both 2-sided p-value and 1-sided p-value are used at the given significant level σ. For the 2-sided p-value, the null hypothesis H_0 is that $F(x)$ and $G(y)$ do not differ with each other significantly and the alternative hypothesis H_1 is that $F(x)$ and $G(y)$ are different significantly, if $p \geq \sigma$, then the null hypothesis H_0 is accepted, otherwise, the alternative hypothesis H_1 is accepted. For 1-sided p-value, there are two cases: in the 1-sided p-value (upper) case, the null hypothesis H_0 is that $F(x)$ does not significantly tends to be greater than $G(y)$ and the alternative hypothesis H_1 is that $F(x)$ significantly tends to be greater than $G(y)$, if $p \geq \sigma$, then H_0 is accepted, otherwise, H_1 is accepted; in the 1-sided (lower) case, the null hypothesis H_0 is that $F(x)$ does not significantly tends to be less than $G(y)$ and the alternative hypothesis H_1 is that $F(x)$ significantly tends to be less than $G(y)$, if $p \geq \sigma$, then H_0 is accepted, otherwise, H_1 is accepted.

Hence, three paired Wilcoxon-Signed-Rank tests are performed for each of the three MRs in our experiment: FP-MS versus FP-ST, FP-MS versus FP-FT and FP-MS versus FP-AT. As a reminder, the group of data of $F(x)$ is the group of Silhouette Coefficients for the proximity matrix in FP-MS, correspondingly, the group of data of $G(y)$ is the group of Silhouette Coefficients for the proximity matrix in FP-ST, FP-FT and FP-AT respectively. Each test uses both 2-sided and 1-sided (upper and lower) checking at the significant level $\sigma = 0.05$.

The results of statistical comparisons are shown in Table I. Taking for example the last row in MR3, for FP-MS versus FP-AT, in the 2-sided test, the p-value is 2.11E-3 which is far less than 0.05, hence, the alternative hypothesis H_1 that the quality of proximity matrix in FP-MS significantly tends to be different from that in FP-AT is accepted; in the 1-sided (upper) test, the p-value is 1.05E-3 which is far less than 0.05, hence, the alternative hypothesis H_1 that the quality of proximity matrix in FP-MS significantly tends to be better than that in FP-AT is accepted; in the 1-sided (lower) test, the p-value is 0.999 which is far larger than 0.05, hence, the null hypothesis H_0 that the quality of proximity matrix in FP-MS do not significantly tends to be worse than that in FP-AT is accepted. In summary, we conclude that the quality of proximity matrix in FP-MS significantly tends to be better than that in FP-AT.

Table I
STATISTICAL COMPARISONS CONCLUSION FOR MR1, MR2 AND MR3

MRs	Scenario	2-sided	1-sided (upper)	1-sided (lower)	Conclusion
MR1	FP-MS vs. FP-ST	0.07099	0.03549	0.96602	Better
	FP-MS vs. FP-FT	0.58799	0.71279	0.294	Similar
	FP-MS vs. FP-AT	3.81E-6	1.91E-6	1	Better
MR2	FP-MS vs. FP-ST	0.83624	0.59311	0.41812	Similar
	FP-MS vs. FP-FT	0.92267	0.46133	0.54932	Similar
	FP-MS vs. FP-AT	5.49E-4	2.75E-4	0.99976	Better
MR3	FP-MS vs. FP-ST	3.05E-5	1	1.53E-5	Worse
	FP-MS vs. FP-FT	1.56E-3	7.78E-4	0.999	Better
	FP-MS vs. FP-AT	2.11E-3	1.05E-3	0.999	Better

From Table I, we observe that in most conditions, the

quality of proximity matrix in FP-MS is similar to or better than the other three scenarios with test oracle. Only in FP-ST for MR3, the Worse condition occurs. Overall, the conditions of Similar and Better account for 89% of the whole results. Consequently, the quality of proximity matrix computed with metamorphic slice is statistical comparable to those scenarios where the test oracle exists.

B. Proximity graphs

Proximity graphs provide a direct visualization of failure proximity to complement the above results of Silhouette Coefficients. Through proximity graphs, we have an intuitive picture of the proximity matrix. In our experiment, for the three MRs, in each scenario, for each mutant under a set of prepared violated metamorphic test groups, the proximity graphs are all computed. In total, we can have 200 proximity graphs. For the limitation of space, we do not display all the proximity graphs. We take a mutant version as an example to illustrate the proximity graphs. The detailed information about the mutant version (denoted as mu_1) is described in Table II. As a reminder, the violated metamorphic test groups in G_1 and G_2 are due to Fault 1 and Fault 2 respectively and the intersection between G_1 and G_2 is empty; Fault 1 and Fault 2 are from two randomly generated single-fault mutants respectively which are introduced in Section IV, where Fault 1 is an off-by-one bug and Fault 2 is an incorrect arithmetic operation.

Table II
CHARACTERISTICS FOR THE EXAMPLE MUTANT VERSION

Mutant	MT group	Fault 1	Fault 2	Violated MT group	$\|G_1\|$	$\|G_2\|$
mu_1	2982	Off-by -one	Incorrect-arith metic-operation	552	419	133

The proximity graphs of mutant version mu_1 for MR1 are shown in Fig. 2. The proximity graphs are obtained through MDS techniques based on proximity matrixes in the four scenarios FP-MS, FP-ST, FP-FT and FP-AT. In Fig. 2, we also give the corresponding Silhouette Coefficients for the four proximity matrixes. From Fig. 2, subfigures (a), (b), (c), (d) represent proximity graphs for FP-MS, FP-ST, FP-FT and FP-AT, respectively. In each proximity graph, there are two shapes crosses and squares which represent violated metamorphic test groups due to Fault 1 and Fault 2 respectively. In the optimal condition (given a Silhouette Coefficient equal to 1), crosses and squares should be separated from each other explicitly but themselves be densely aggregated. In Fig. 2a (FP-MS), all squares collapse together and crosses are scattered, while in Fig. 2b (FP-ST) all squares form two dense clusters and crosses have similar scattering as that in Fig. 2a, thus, we can make a judgement visually that FP-MS of Fig. 2a (0.87763) is likely to have better proximity quality than FP-ST of Fig. 2b (0.79824). Similarly, in Fig. 2d (FP-AT), we observe that all squares

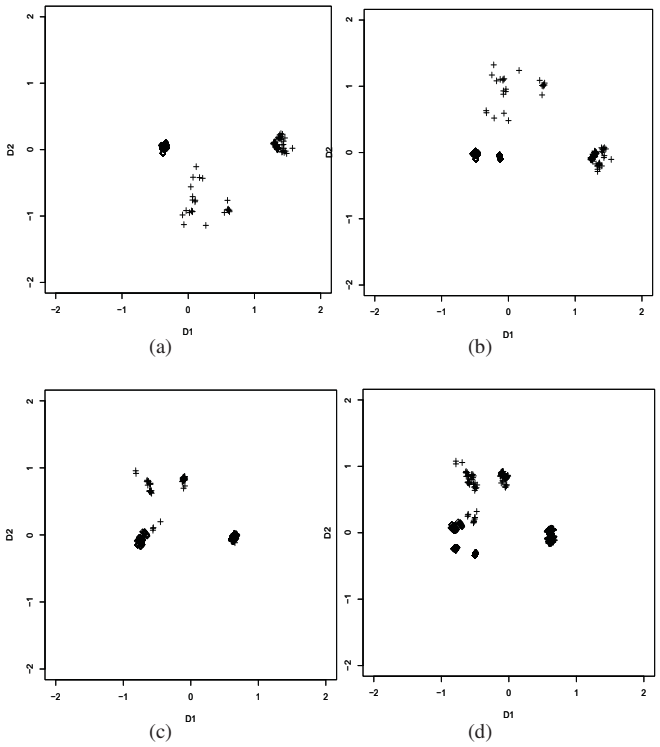


Figure 2. Comparison of proximity graphs of mu_1 for MR1. (a) FP-MS(0.87763). (b) FP-ST (0.79824) (c) FP-FT (0.94408). (d) FP-AT (0.72872)

form three dense clusters and crosses are more scattered than that in Fig. 2a, thus FP-MS of Fig. 2a (0.87763) is likely to have better proximity quality than FP-AT of Fig. 2d (0.72872). Among the four proximity matrixes, FP-FT has the highest Silhouette Coefficient 0.94408. In Fig. 2c (FP-FT), all squares collapse together and crosses form several dense clusters which is more cohesive than that in Fig. 2a (0.87763). Thus, the quality of proximity graph in Fig. 2a may be worse than that in Fig. 2c.

C. Threats to validity

In this part, three kinds of threats to validity are considered in interpreting the experiment results. The threat to external validity may exist when our results are generalized to arbitrary program. First, the subject program *grep* used in our experiment is not a very large-sized program. Thus, in the further work of our approach, investigation of more large-sized programs would strengthen the validity of our approach. Second, the identification of effective MRs is not investigated in this study. Recent work provided empirical evidence that a small number of MRs are able to effectively help alleviate the test oracle problem [11, 22]. The three MRs defined in our study are widely understood and commonly used in both academic and industrial community, thus, the threat of effective MRs is acceptable mitigated.

The threat to internal validity includes the assumption that a violated metamorphic test group is due to one and only one fault. However, in practice the violation can be caused by multiple faults. However, it is highly debatable to determine whether a violated metamorphic test group is caused by one fault or multiple faults. In the further work, research on this problem would be necessary.

The threat to construct validity is the use of Silhouette Coefficient and proximity graphs as the metrics to evaluate the quality of proximity matrix with test oracle exists or not. Although the two metrics may not conform to the realistic human evaluation in all aspects, from the previous work of Liu et al. [2], the design of the two metrics intentionally avoid subjectiveness, hence, the quality of proximity matrixes can be objectively evaluated.

VI. Related work

Metamorphic testing has been successfully applied to various application domains in order to detect the inherent faults. Such as in machine learning application where the test oracle is difficult to obtain, Xie et al. [8, 23] applied MT to test and validate classifiers like KNN and Naïve Bayes classifier in Weka and some potential faults in these programs are detected. Also, real-life faults are detected in bioinformatics programs [9], C compilers [24], etc.

Besides, metamorphic testing has been integrated with other software analysis and testing techniques. Gotlieb developed an automated testing framework through integrating metamorphic testing with constraint logical programming techniques [25]. Xie et al. [14] proposed the concept of metamorphic slice based on the integration of metamorphic testing and slicing, and applied metamorphic slice to spectrum-based fault localization (SBFL) without test oracles. The empirical evidence is provided that metamorphic slice is able to help extend the applicability of SBFL to programs without test oracles. In this paper, we focus on how to extend the applicability of code coverage-based failure proximity to programs without test oracles.

VII. Conclusion

Failure indexing techniques are or crucial importance in modern software maintenance. It plays a key important role in facilitating duplicated failures removal, failure assignment, etc. Generally, failure proximity is the crucial part of underpinning an effective failure indexing technique. Hence, Liu et al. [2] investigated several failure proximity approaches for crashing failures and non-crashing failures. No matter the various form of different failure proximity, each of them is an instantiation of a fingerprinting function and a distance function. However, all the failure proximity approaches for non-crashing failures assume the existence of test oracles. In practice, there exist many application domains lacking of test oracles, hence, the extensions of applicability of failure proximity on these programs are limited. Fortunately, recent research work has showed that metamorphic testing is simple in conception and effective to alleviate the test oracle problem [11]. Consequently, in order to extend the applicability of failure proximity to these programs without test oracles, we consider how MT could be applied to alleviate the oracle problem in failure proximity.

In this paper, we focus on investigating code coverage-based failure proximity. A new algorithm is proposed to alleviate the oracle problem through several replacements of components in traditional failure proximity approach. The failures are replaced by violated metamorphic test groups; the failure signature of execution slices is replaced by metamorphic slices in fingerprinting function; the distance between two failures is replaced by the distance between two violated metamorphic test groups in distance function. Thereby, metamorphic slice could be applied to enhance the applicability of code coverage-based failure proximity.

An experimental study is conducted to investigate the effectiveness of our proposed approach. We choose the real-life UNIX utility program *grep* as the subject program and generate two-fault mutants through combining two single-fault mutants which are randomly generated. The quality of failure proximity is measured quantitatively and qualitatively by two metrics of Silhouette Coefficient and proximity graphs respectively. The results show that our approach is statistical comparable to traditional code coverage-based failure proximity for the situations with test oracle. It indicates that our approach is a successful application of metamorphic slice on failure proximity without test oracles.

Future work includes but is not limited in conducting additional experiment on more large-sized programs, investigating more complicated failure proximity approaches without test oracles, as well as programs with more than two faults.

Acknowlogement

This research is partly supported by the National Natural Science Foundation of China (91418202, 61472178 and 61170071), the National Basic Research Program of China (2014CB340702) and Postgraduate Innovation Project of Jiangsu Province (CXZZ13_0054).

References

[1] S. Wang, F. Khomh, and Y. Zou, "Improving bug management using correlations in crash reports," *Empirical Software Engineering*, pp. 1–31, 2014.

[2] C. Liu, X. Zhang, and J. Han, "A systematic study of failure proximity," *IEEE Transactions on Software Engineering*, vol. 34, no. 6, pp. 826–843, 2008.

[3] J. A. Jones, J. F. Bowring, and M. J. Harrold, "Debugging in parallel," in *Proceedings of the 2007 International Symposium on Software Testing and Analysis (ISSTA)*, 2007, pp. 16–26.

[4] W. Dickinson, D. Leon, and A. Podgurski, "Persuing failure: the distribution of program failures in a profile space," in *Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE)*, 2001, pp. 246–255.

[5] A. Podgurski, D. Leon, P. Francis, W. Masri, M. Minch, J. Sun, and B. Wang, "Automated support for classifying software failure reports," in *Proceedings of the 25th International Conference on Software Engineering (ICSE)*, 2003, pp. 465–475.

[6] Y. You, C. Huang, K. Peng, and C. Hsu, "Evaluation and analysis of spectrum-based fault localization with modified similarity coefficients for software debugging," in *Proceedings of 37th Annual International Computer Software and Applications Conference (COMPSAC)*, 2013, pp. 180–189.

[7] C. Liu and J. Han, "Failure proximity: a fault localization-based approach," in *Proceedings of the 14th ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE)*, 2006, pp. 46–56.

[8] X. Xie, J. W. K. Ho, C. Murphy, G. Kaiser, B. Xu, and T. Y. Chen, "Testing and validating machine learning classifiers by metamorphic testing," *Journal of Systems and Software*, vol. 84, pp. 544–558, 2011.

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

[10] 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*, no. 4, pp. 61–81, 2007.

[11] H. Liu, F.-C. Kuo, D. Towey, and T. Y. Chen, "How effectively does metamorphic testing alleviate the oracle problem," *IEEE Transactions on Software Engineering*, vol. 40, no. 1, pp. 4–22, 2014.

[12] F.-C. Kuo, T. Chen, and W. Tam, "Testing embedded software by metamorphic testing: a wireless metering system case study," in *Proceedings of the 36th IEEE International Conference on Local Computer Networks (LCN)*, 2011, pp. 295–298.

[13] S. Beydeda, "Self-metamorphic-testing components," in *Proceedings of 30th Annual International Computer Software and Applications Conference (COMPSAC)*, 2006, pp. 265–272.

[14] X. Xie, W. Wong, T. Chen, and B. Xu, "Metamorphic slice: an application in spectrum-based fault localization," *Information and Software Technology*, vol. 55, no. 5, pp. 866–879, 2013.

[15] Y. Lei, X. Mao, and T. Y. Chen, "Backward-slice-based statistical fault localization without test oracles," in *Proceedings of the 13th International Conference on Quality Software (QSIC)*, 2013, pp. 212–221.

[16] M. Levandowsky and D. Winter, "Distance between sets," *Nature*, vol. 234, pp. 34–35, Nov. 1971.

[17] T. Chen, S. Cheung, and S. Yiu, "Metamorphic testing: a new approach for generating next test cases," Department of Computer Science, Hong Kong University of Science and Technology, Tech. Rep. HKsUST-CS98-01, 1998.

[18] C. Murphy, K. Shen, and G. Kaiser, "Automatic system testing of programs without test oracles," in *Proceedings of 18th ACM International Symposium on Software Testing and Analysis (ISSTA)*, 2009, pp. 189–200.

[19] SIR, http://sir.unl.edu/php/index.php, 2005.

[20] I. Borg and P. Groenen, *Modern multidimensional scaling: theory and applications*, 1st ed. Springer, 1996.

[21] G. Corder and D. Foreman, *Nonparametric statistics for non-statisticians: a step-by-step approach*. Wiley, 2009.

[22] Z. Zhang, W. Chan, T. Tse, and P. Hu, "Experimental study to compare the use of metamorphic testing and assertion checking," *Journal of Software*, vol. 20, pp. 2637–2654, 2009.

[23] X. Xie, J. Ho, C. Murphy, G. Kaiser, B. Xu, and T. Chen, "Application of metamorphic testing to supervised classifiers," in *Proceedings of the 9th International Conference on Quality Software (QSIC)*, 2009, pp. 135–144.

[24] C. Z. Q. Tao, W. Wu and W. Shen, "An automatic testing approach for compiler based on metamorphic testing technique," in *Proceedings of 17th Asia Pacific Software Engeering Conference (APSEC)*, 2010, pp. 270–279.

[25] A. Gotlieb and B. Botella, "Automated metamorphic testing," in *Proceedings of 27th Annual International Computer Software and Applications Conference (COMPSAC)*, 2003, pp. 34–40.