2022 IEEE International Conference on Software Maintenance and Evolution (ICSME) | 978-1-6654-7956-1/22/$31.00 ©2022 IEEE | DOI: 10.1109/ICSME55016.2022.00026

Improving Fault Localization Using Model-domain Synthesized Failing Test Generation

Zhuo Zhang[1,2], Yan Lei[1,5*], Xiaoguang Mao[3], Meng Yan[1], Xin Xia[4]
[1]School of Big Data & Software Engineering, Chongqing University, Chongqing, China
[2]Guangzhou College of Commerce, Guangzhou, China.
[3]College of Computer, National University of Defense Technology, Changsha, China
[4]Software Engineering Application Technology Lab at Huawei, China
[5]Peng Cheng Laboratory, Shenzhen, China
zz8477@126.com, yanlei@cqu.edu.cn, xgmao@nudt.edu.cn, mengy@cqu.edu.cn, xin.xia@acm.org

***Abstract*—A test suite is indispensable for conducting effective fault localization, and has two classes of tests: passing tests and failing tests. However, in practice, passing tests heavily outnumber failing tests regarding a fault, leading to failing tests being a minority class in contrast to passing tests. Previous work has empirically shown that the lack of failing tests regarding a fault leads to a class-balanced test suite, which tends to hamper fault localization effectiveness.**

To address this issue, we propose MSGen: a Model-domain Synthesized Failing Test Generation approach. MSGen utilizes the widely used information model of fault localization (*i.e.,* an abstraction of the execution information and test results of a test suite), and uses the minimum variability of the minority feature space to create new synthesized model-domain failing test samples (*i.e.,* synthesized vectors with failing labels defined as the information model) for fault localization. In contrast to traditional test generation directly from the input domain, MSGen seeks to synthesize failing test samples from the model domain. We apply MSGen to 12 state-of-the-art localization approaches and also compare MSGen to 2 representative data optimization approaches. The experimental results show that our synthesized test generation approach significantly improves fault localization effectiveness with up to 51.22%.

***Index Terms*—fault localization; synthesized test generation; model domain; suspiciousness**

I. Introduction

In order to reduce the software debugging cost [1], researchers have developed many fault localization techniques to provide the assistance in seeking the positions of the faults in the program (*e.g.,* [2]–[8]). Among them, spectrum-based fault localization (SBFL) [9], [10] and deep-learning-based fault localization (DLFL) [11]–[14] are the two of the most popular ones showing promising results [15], [16].

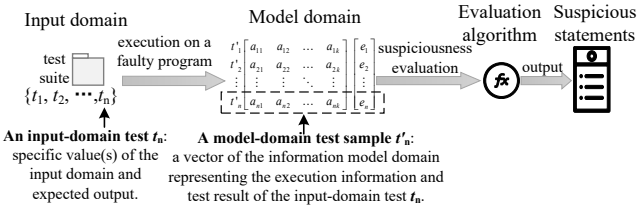


Fig. 1. A typical fault localization process of SBFL and DLFL.

Given a faulty program $\mathcal{P}$ containing a fault, Fig. 1 shows a typical process of SBFL and DLFL as follows:

Execution of input-domain tests. They require a collection of input-domain tests to construct a test suite $\mathcal{T}$, and execute $\mathcal{T}$ on the program $\mathcal{P}$. An input-domain test consists of specific value(s) of the input domain of the program $\mathcal{P}$ and the expected output.

Construction of model-domain test samples. Then, they collect and abstract the coverage information (*i.e.,* a statement *executed* or *not executed*) and test result (*i.e., passing* or *failing*) of each input-domain test as a model-domain test sample. All model-domain test samples form an information model (usually denoted as a matrix) [10], [11], [17], [18] for fault localization algorithms. Specifically, a model-domain test sample is a vector of the information model domain representing the coverage information (*i.e., executed* or *not executed*) of each statement and the test result (*i.e., passing* or *failing*) of an input-domain test (see Section II-A for more detail).

Suspiciousness evaluation. Finally, based on the information model (*i.e.,* model-domain tests), they use an evaluation algorithm (*e.g.,* SBFL using statistical correlation coefficients [10], [16] or DLFL using neural networks [11], [19]) to evaluate the suspiciousness of each statement (or other program elements) of being faulty and rank all the statements in descending of suspiciousness.

The test suite $\mathcal{T}$ is a vital component to initiate the fault localization process, and has two classes of input-domain tests with a distinct feature: failing tests and passing ones. As shown in Fig. 1, the state-of-the-art fault localization techniques usually use either statistical correlation coefficients (*e.g.,* SBFL) or neural networks (*e.g.,* DLFL) and thus require an adequate number of tests, including passing ones and failing ones. However, regarding a fault, the number of failing tests is usually much fewer than passing tests in practice, leading to a class imbalanced problem which may pose a threat to fault localization effectiveness. Therefore, the prior studies [20], [21] have evaluated the impact of class imbalance in a test suite on fault localization, and their results have shown that class imbalance hampers fault localization effectiveness.

Regarding a fault, it is difficult to generate failing tests directly from the input domain [22]–[27] because (1) those inputs causing program failures via this fault generally account

*Yan Lei is the corresponding author.

2576-3148/22/$31.00 ©2022 IEEE
DOI 10.1109/ICSME55016.2022.00026

for a very small portion of the input domain, and (2) their distribution is usually sporadic and even random. Thus, although researchers have spent much effort in generating tests for fault localization, the existing test generation approaches (*e.g.,* [20], [28], [29]) rarely generate failing tests for fault localization, but instead, they optimize or generate passing tests for fault localization. Even worse, the existing studies [12], [21], [30]–[32] have found that regarding a fault, failing tests are always beneficial for fault localization and the class imbalance problem will jeopardize fault localization due to a bias to passing tests. Therefore, there is an urgent need to tackle the class imbalance problem for improving fault localization effectiveness.

Since generating failing test cases directly from the input domain is difficult, we seek a different perspective to address the class imbalance problem. As shown in Fig. 1, fault localization usually abstracts the statement coverage and test results of all input-domain tests into an information model [10], [11] (*i.e.,* model-domain test samples) to represent the program behavior. In other words, a model-domain test sample is composed of the coverage of each statement (*i.e., executed* or *not executed*) and the test results (*i.e., passing* or *failing*), and thus we can treat the statements and the test results as features and labels in the machine learning domain. In machine learning, data synthesis is a commonly used solution to the problem of class-imbalanced data [33], and many studies (*e.g.,* [34]–[37]) have shown that creating synthesized minority class samples can improve the accuracy of the models. Inspired by data synthesis, we intend to create new synthesized model-domain failing test samples by extracting and keeping common features (*i.e.,* specific statements) from existing model-domain failing test samples in information model domain. We expect the synthesized model-domain failing test samples could help to improve the effectiveness of SBFL and DLFL.

Based on the above observation, we propose MSGen: a Model-domain Synthesized Failing Test Generation approach using the minimum variability of the failing class feature space (*i.e.,* preserving the common features) to synthesize model-domain failing test samples (*i.e.,* vectors with failing labels) for improving fault localization. MSGen adopts the widely used information model [10] in fault localization, where a vector records the execution information and test result of an input-domain test showing that what statements are *executed* (or *not executed*) by the test with a *passing* (or *failing*) result (see Section II-A for more detail). Inspired by data synthesis techniques [37], MSGen utilizes the existing model-domain failing test samples to create new synthesized model-domain failing test ones by covering the minimum suspicious set. Specifically, for a vector of a model-domain failing test sample, MSGen computes its nearest neighbor from existing model-domain failing test samples, and their intersection (*i.e.,* common features) covers those statements executed by all failing tests (*i.e.,* minimum suspicious set [30]). For preserving the common features, MSGen only varies the subtraction (*i.e.,* difference) of the model-domain failing test sample from its nearest neighbor to produce a new synthesized model-domain failing test sample (*i.e.,* a new vector from the information model with a failing label). Thus, the new synthesized model-domain failing test sample covers minimum suspicious set and should be beneficial for improving fault localization. Finally, MSGen iteratively creates model-domain failing test samples until obtaining a balanced test suite, where model-domain failing test samples and model-domain passing test ones have the same size in the information model as shown in Fig. 2.

In comparison to traditional test generation from the input domain, model-domain synthesized failing test generation is easier to synthesize new failing vectors from the model domain and does not need the execution of a test to fix its result label. Thus, a synthesized model-domain failing test sample does not necessarily correspond to a real piece of data from the input domain to execute such a path denoted by the model-domain test sample.

To evaluate our approach, we apply MSGen to 12 state-of-the-art fault localization approaches [9], [11]–[16] and compare MSGen with two representative data optimization approaches [21], [38]. The large-scale empirical study on 14 real-life programs shows that MSGen significantly outperforms the 12 localization approaches and two data optimization approaches in terms of fault localization effectiveness.

The main contributions of this paper can be summarized as:

- We propose a fresh perspective of producing synthesized test samples from the model domain, rather than generating real tests from the input domain, to improve the accuracy of fault localization algorithms.
- We propose a model-domain synthesized failing test generation approach using the minimum variability of the failing class feature space to synthesize new model-domain failing test samples for fault localization.
- We conduct an experimental study on 14 large real-life programs, showing that MSGen is effective to improve fault localization.

The structure of the rest paper is organized as follows. Section II introduces the background on fault localization. Section III describes our model-domain synthesized failing test generation approach MSGen. Section IV presents our large-scale empirical study. Section VI summarizes related work and Section VII concludes.

II. Background

This section will introduce the widely used information model in fault localization, and the 12 state-of-the-art fault localization approaches used in the experiments.

A. Information Model

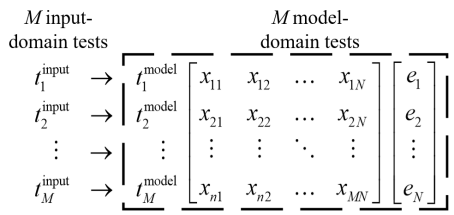


Fig. 2. The definition of the information model.

Fault localization usually defines an information model, *i.e.,* a representation of the coverage information and test results of a test suite (*i.e.,* a set of input-domain tests) on a faulty program, to provide the information for initiating and executing localization algorithms to locate faults. In fault localization, the widely used information model is program spectra model [10].

Fig. 2 shows the definition of program spectra model. Specifically, given a program *P* with *N* statements, it is executed by a test suite *T* with *M* input-domain tests, *i.e.,* $T = \{t_1^{input}, t_2^{input}, ..., t_M^{input}\}$ (see Fig. 2). Here, an input-domain test t_i^{input} is defined as follows:

> **An input-domain test t_i^{input}:** a test input (*i.e.,* test data from input domain), a test oracle (*i.e.,* expected output) and others.

We execute each input-domain test to collect its coverage information (*executed* or *unexecuted* statements) and check the output against its test oracle (*passing* or *failing* test result). Fault localization techniques define and construct a model-domain test sample to represent the statement coverage information and test result of each input-domain test. Here, a model-domain test sample t_i^{model} is defined as follows:

> **A model-domain test sample** t_i^{model}: a vector with $(N + 1)$ elements denoted as $[x_{i1}, x_{i2}, ..., x_{iN}, e_i]$, recording the coverage information of each statement and the test result of the input-domain test t_i^{input}.

Where, the element x_{ij}=1 means that the statement j is executed by the input-domain test t_i^{input}, and x_{ij}=0 otherwise. The element e_i equals to 1 if the input-domain test t_i^{input} failed, and 0 otherwise. The M elements of e_i ($i \in \{1, 2.., M\}$) are usually denoted as an error vector $e = [e_1, e_2, ..., e_M], i \in \{1, 2.., M\}$.

The *M* model-domain test samples form the information model (*i.e.,* a *M*×(*N+1*) matrix) recording the coverage information (*executed* or *unexecuted*) of each statement in the test suite *T*, and the test results (*i.e., passing* or *failing*) of each input-domain test. Thus, an information model is defined as follows:

> **An information model:** an *M*×(*N+1*) matrix, where the *i*-th row is the model-domain test sample $t_i^{model} = [x_{i1}, x_{i2}, ..., x_{iN}, e_i]$.

Based on the information model, fault localization designs the localization algorithms (*e.g.,* neural networks [11], [13]–[15], [19] and statistical correlation coefficients [10], [16]) to evaluate the suspiciousness of each statement being faulty.

B. Deep-Learning-based Fault Localization

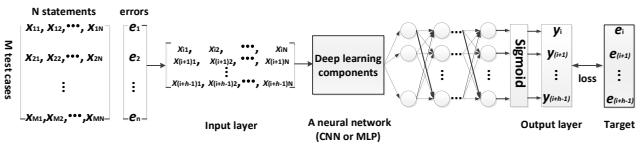


Fig. 3. The architecture of Deep-Learning-based fault localization.

Deep-learning-based fault localization (DLFL) utilizes the promising learning ability [39]–[43] of a neural network to learn a fault localization model to evaluate the suspiciousness of each statement being faulty. Among these DLFL techniques, MLP-FL [19], CNN-FL [11], BiLSTM-FL [12], FLUCCS [14] and DeepFL [13] are the representative and effective ones.

For simplicity, we will depict two DLFL localization (*i.e.,* MLP-FL and CNN-FL) approaches as the representative. Fig. 3 shows their architecture: one input layer, deep leaning components, several hidden layers, and one output layer. In the input layer, DLFL takes the information model defined in Fig. 2 as input, *i.e., h* rows of the matrix *M*×*N* and its corresponding error vector are used as an input. In deep learning components, there may be convolution layers, pooling layers or fully connected layers. After that, there may be some hidden layers. CNN-FL and MLP-FL use convolutional neural network and multi-layer perceptron for deep learning components respectively. In the output layer, the models use *sigmoid* function [11] because values sent into a *sigmoid* function will be 0 to 1. Each element in the result vector of the sigmoid function has difference with the corresponding element of the target vector. Back propagation algorithm is used to fine-tune the parameters of the model, and the goal is to minimize the difference between training result *y* and error vector *e*. The network is trained iteratively.

The training process will learn a trained model, reflecting the complex nonlinear relationship between the statement coverage and test results. Finally, the model constructs a set of virtual tests as the testing input to measure the association of each statement with test results. Since each virtual test only covers one statement, each time the model inputs one virtual test to the network and the output is the estimation of the probability of causing a failure by only executing the statement. Thus, the estimation is the suspiciousness of a statement of being faulty.

C. Spectrum-based Fault Localization

Spectrum-based Fault Localization (SBFL) [10] also uses the information model in Fig. 2, and defines the following four variables.

$$\begin{aligned} a_{np}(s_j) &= |\{i|x_{ij} = 0 \wedge e_i = 0\}|, \quad a_{nf}(s_j) = |\{i|x_{ij} = 0 \wedge e_i = 1\}| \\ a_{ep}(s_j) &= |\{i|x_{ij} = 1 \wedge e_i = 0\}|, \quad a_{ef}(s_j) = |\{i|x_{ij} = 1 \wedge e_i = 1\}| \end{aligned} \tag{1}$$

Eq. 1 shows the computation of a_{np}, a_{nf}, a_{ep}, and a_{ef} for the statement j (*i.e.,* s_j), denoting the number of passing/failing tests in which the statement was/wasn't executed[1].

With the four variables for each statement, SBFL defines many suspiciousness evaluation formulas to evaluate the suspiciousness of each statement being faulty. Researchers have conducted both theoretical [9], [44] and empirical [16] analysis on finding the optimal SBFL formulas, *i.e.,* ER1', ER5, GP02, GP03, GP19, Dstar and Ochiai.

[1]If the values of the elements of a vector are a decimal number, the meaning of the four variables are slightly different (*e.g.,* [17], [18]).

III. SYNTHESIZED TEST GENERATION

A. Overview

We propose MSGen to synthesize model-domain failing test samples (*i.e.,* synthesized vectors with failing labels) from the model domain, rather than from the input domain, to improve fault localization. A synthesized model-domain failing test sample is a vector with a failing label and the same structure of the information model defined in Fig. 2. In other words, its elements and failing label correspond to the definition of x_{ij} and e_i in Fig. 2, respectively.

Data synthesis The previous research [30] has identified the data covering the features of all failing tests (*i.e.,* minimum suspicious set) are beneficial (or safe) for fault localization. The minimum suspicious set is defined as the set of these statements executed by all failing tests. It is intuitive that a faulty statement should be executed to cause a failure and the faulty statement should be in the minimum suspicious set. Fault localization will increase the suspiciousness of the faulty statement if the faulty statement is in those vectors with a failing label. Since these new vectors produced by covering the minimum suspicious set should include the faulty statement, fixing the failing label to those vectors is beneficial for increasing the suspiciousness of the faulty statement.

Thus, MSGen seeks to use the information of existing model-domain failing test samples (*i.e.,* those vectors with a failing label) to synthesize new failing vectors covering the minimum suspicious set for fault localization. Specifically, for each model-domain failing test sample, MSGen uses synthetic minority over-sampling technique [45] to synthesize its *k* nearest neighbors from the other model-domain failing test samples, where the nearest neighbor is also a model-domain failing test sample. Thus, the intersection (*i.e.,* common features) of the model-domain failing test sample and its nearest neighbors covers the minimum suspicious set. To preserve the common features, MSGen computes the subtraction (*i.e.,* difference) of the model-domain failing test sample from its nearest vector to generate a new vector. Since the new vector covers the common features, MSGen takes the new vector as a new model-domain failing test sample and adds the new synthesized model-domain failing sample into the original information model.

Data quantity MSGen should identify the amount of synthesized model-domain failing test samples to be created. Many studies have found that a class-balanced test suite is useful for fault localization [12], [21], and algorithms with balanced data should generally surpass those with imbalanced data in performance [31], [32]. Therefore, MSGen produces synthesized model-domain failing test samples until we obtain a balanced test suite, in which the number of model-domain passing test samples and model-domain failing test ones are the same. It means that the vectors with a failing label have the same number of those vectors with a passing label.

B. Methodology

Algorithm 1 depicts the algorithm of MSGen. Suppose that there are at least 2 failing test cases in the original test suite.

Algorithm 1 The MSGen Algorithm

Input: The matrix of the original test suite, $TOrig$; The number of nearest neighbors, k;
Output: The matrix of the new test suite, $TNew$.

1: $TNew$=$TOrig$;
2: $TOrigF$=getFailingTests($TOrig$);
3: $Pnum$=getNumberofPassingTests($TOrig$);
4: $Fnum$=getNumberofFailingTests($TOrig$;
5: **for** $i = 1$; $i <= Fnum$; i++ **do**
6: Compute k nearest neighbors for $TorigF[i]$ from the set of the other model-domain failing test samples (*i.e.,* $TorigF - TorigF[i]$), and save the indices in the $nnarray[i]$.
7: **end for**
8: $FNewnum = Pnum - Fnum$;
9: $index = 1$;
10: **for** $i = 1$; $i <= FNewnum$; i++ **do**
11: **if** $index\%(Fnum + 1) == 0$ **then**
12: $index = 1$;
13: **end if**
14: Choose a random number between 1 and k, call it nn. This step chooses one of the k nearest neighbors of $TOrigF[index]$.
15: $t_{nearest} = TOrigF[nnarray[index][nn]]$;
16: $t_{new} = TOrigF[index] + rand(0, 1) * |t_{nearest} - TOrigF[index]|$;
17: Fix a *failing* label to t_{new}.
18: add($TNew$,t_{new});
19: $index++$;
20: **end for**
21: **return** T_{New};

For the input and output of Algorithm 1, the matrix of a test suite is the information model (*i.e.,* model-domain test samples) defined in Fig. 2, showing the statement coverage and test results. Line 1-4 initialize $TNew$ (*i.e.,* the output), $TOrigF$ (*i.e.,* the original model-domain failing test samples), $Pnum$ and $Fnum$ (*i.e.,* the number of model-domain passing and failing test samples from the original test suite).

To cover the minimum suspicious set, lines 5-7 compute the k-nearest neighbors of each model-domain failing test sample. Specifically, for each model-domain failing test sample denoted as a vector in Fig. 2, MSGen calculates its Euclidean distance [45] to all the other model-domain failing test samples and selects k model-domain failing test samples with the shortest Euclidean distance as its k-nearest neighbors. We can observe that a model-domain failing test sample and its nearest neighbors are the model-domain failing test samples from the original test suite, and the minimum suspicious set is defined as the set of these statements executed by all model-domain failing test samples from the original test suite. Thus, the model-domain failing test sample and its nearest neighbors cover the minimum suspicious set.

Lines 8-20 repeat the production of new synthesized model-domain failing test samples until the model-domain failing test

TABLE I
SUMMARY OF SUBJECT PROGRAMS.

Program	Description	Versions	KLOC	Test	Coverage	Type
chart	JFreeChart	26	96	2205	59%	Real
math	Apache Commons Math	106	85	3602	78%	Real
lang	Apache commons-lang	65	22	2245	26%	Real
closure	Closure Compiler	133	90	7927	77%	Real
mockito	Framework for unit tests	38	6	1075	71%	Real
time	Joda-Time	27	53	4130	75%	Real
python	General-purpose language	8	407	355	16%	Real
gzip	Data compression	5	491	12	39%	Real
libtiff	Image processing	12	77	78	59%	Real
space	ADL interpreter	38	6.1	13585	100%	Real
nanoxml_v1	XML parser	7	5.4	206	76%	Seeded
nanoxml_v2	XML parser	7	5.7	206	72%	Seeded
nanoxml_v3	XML parser	10	8.4	206	78%	Seeded
nanoxml_v5	XML parser	7	8.8	206	78%	Seeded

samples have the same number as the model-domain passing test samples. For each iteration, MSGen selects a neighbor from the *k* nearest neighbors of the model-domain failing test sample $TOrigF[index]$, and uses the neighbor to synthesize a new model-domain failing test sample by using the equation at Line 16. The equation at the Line 16 uses the subtraction (*i.e.,* difference) of $TOrigF[index]$ from its selected neighbor to synthesize a new model-domain failing test sample. It means that the new synthesized model-domain failing test sample covers the minimum suspicious set since $TOrigF[index]$ and its neighbor are both model-domain failing test samples from original test suite and the minimum suspicious set is the intersection of all model-domain failing test samples. Thus, MSGen fixes a failing label to the synthesized new model-domain test sample at Line 17. Note that MSGen does not care whether there exists data from input domain corresponding to those synthesized model-domain failing test samples (*i.e.,* synthesized vectors with a failing label). For a synthesized model-domain failing test sample, it requires no actual execution and we may not have a real piece of data from input domain to execute such path denoted by the model-domain failing test sample. As a reminder, if there is only one failing test case, we clone the failing test case repeatedly until the failing test cases have the same number as the passing ones.

Finally, MSGen feeds the new matrix (*i.e.,* T_{New}) with the same size of model-domain failing test samples and model-domain passing test ones for fault localization. DLFL and SBFL take as input the new matrix and conduct the evaluation of the suspiciousness of each statement being faulty. They output a ranking list of all statements in descending order of suspiciousness.

IV. AN EXPERIMENTAL STUDY

A. Experimental Setup

Benchmarks The experiments choose the subject programs fro the three reasons: (1) they are the widely used large-sized programs (*e.g.,* [1], [10]–[13], [16]) in fault localization; (2) they are large-sized programs at least more than 5 KLOC; (3) they are easy to be acquired for enabling comparable and reproducible studies. Table I summarizes the characteristics of the subject programs, listing functional description, the number of faulty versions, the number of thousand lines of statements, the number of test cases of the test suite, the statement coverage of the test suite, and the type of the faults. The first six programs (*i.e., chart*, *math*, *lang*,*closure*,*mockito*, and *time*) are from Defects4J[2]. The *python*, *gzip* and *libtiff* are collected from ManyBugs[3]. The *space* and the four separated releases of *nanoxml* are acquired from the SIR[4].

Baselines Prior studies [9], [16], [44] have conducted theoretical [9], [44] and empirical analysis [16] on finding the optimal SBFL formulas, *i.e.,* Ochiai, ER5, GP02, GP03, Dstar, GP19 and ER1'. Furthermore, the recent results [11]–[14] on DLFL have identified five representative and effective ones, *i.e.,* MLP-FL, CNN-FL, BiLSTM-FL, FLUCCS and DeepFL. Therefore, the experiments use the 12 state-of-the-art fault localization approaches as the baselines to evaluate the effectiveness in two scenarios: using our approach MSGen and without using it. Furthermore, we utilize 2 representative and effective data optimization approaches [12], [21], [38] (denoted as undersampling and resampling) for improving fault localization, where one [38] uses undersampling by removing the majority class samples and the other one [12], [21] utilizes resampling by replicating minority class. As a reminder, since the prior study [30] has shown that a test with a higher overlap with minimal suspiscious statements is more beneficial for fault localization, the resampling used by our study generates tests covering all minimal suspicious statements.

For seven SBFL techniques, we implement them based on the widely used SBFL source code GZoltar[5]; for five DLFL techniques, we implement them based on the source code from the previous studies [11]–[15].

Environment The physical environment of the experiments is on a computer containing a CPU of Intel I5-2640 with 128G physical memory, and two 12G GPUs of NVIDIA TITAN X Pascal. The operating system is Ubuntu 16.04.3. We conducted the experiments on the MATLAB R2016b.

B. Evaluation Metrics

To evaluate fault localization effectiveness, we adopt four widely used metrics: *Top-N* [2], *Mean Average Rank (MAR)* [13], *Mean First Rank (MFR)* [13], and *Relative Improvement (RImp)* [46]–[48].

Top-N It denotes the percentage of faults located within the first *N* position of a ranked list of all statements in descending order of suspiciousness returned by a localization approach.

Mean Average Rank (MAR) It is the mean of the average rank of all faults using a localization approach.

Mean First Rank (MFR) For a fault with multiple faulty statements, locating the first one is critical since the others may be located after that. *MFR* is the mean of the first faulty statement's rank of all faults using a localization approach.

Relative Improvement (RImp) It is to compare the total number of statements that need to be examined to find all

[2]Defects4J, http://defects4j.org
[3]ManyBugs, http://repairbenchmarks.cs.umass.edu/ManyBugs/
[4]SIR, http://sir.unl.edu/portal/index.php
[5]https://gzoltar.com/

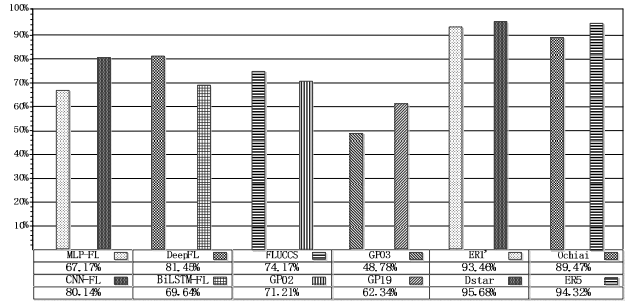
faults using a fault localization approach with MSGen versus the number that need to be examined by using the one without MSGen or with a different data optimization approach.

C. Localization with MSGen versus without MSGen

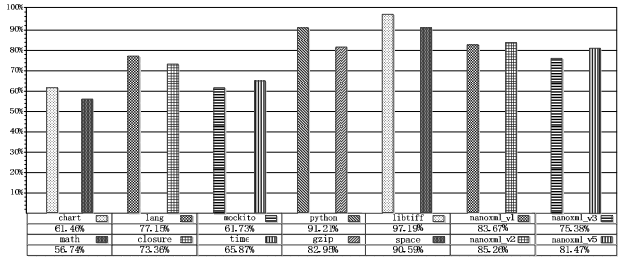
TABLE II
Top-N, *MAR* AND *MFR* COMPARISON OF MSGEN OVER THE 12 FAULT LOCALIZATION APPROACHES.

Comparison	top-1	top-5	top-10	top-20	MFR	MAR
MLP-FL	0%	2.1%	3.1%	12.4%	213	352
MLP-FL(MSGen)	**1.6%**	**5.7%**	**11.9%**	**19.1%**	**194**	**337**
CNN-FL	1.6%	3.1%	8.2%	18.0%	179	263
CNN-FL(MSGen)	**1.6%**	**9.3%**	**16.5%**	**26.3%**	**122**	**248**
BiLSTM-FL	0%	1.6%	3.1%	10.8%	235	412
BiLSTM-FL(MSGen)	**0%**	**3.6%**	**9.3%**	**17.0%**	**207**	**369**
DeepFL	1.6%	4.1%	9.3%	21.1%	188	271
DeepFL(MSGen)	**1.6%**	**9.3%**	**17.0%**	**25.8%**	**131**	**239**
FLUCCS	0%	6.2%	9.3%	17.0%	193	321
FLUCCS(MSGen)	**1.6%**	**9.3%**	**17.0%**	**25.8%**	**159**	**288**
ER5	0%	6.2%	12.4%	20.6%	247	421
ER5(MSGen)	**0%**	**7.7%**	**14.4%**	**27.3%**	**215**	**384**
GP02	1.6%	15.5%	20.6%	37.1%	263	545
GP02(MSGen)	**2.6%**	**17.0%**	**23.2%**	**40.7%**	**221**	**476**
GP03	3.1%	11.3%	12.4%	19.1%	217	363
GP03(MSGen)	**3.1%**	**15.5%**	**18.0%**	**21.1%**	**197**	**326**
Dstar	3.1%	20.1%	26.3%	40.2%	241	357
Dstar(MSGen)	**3.6%**	**23.2%**	**31.4%**	**46.9%**	**219**	**317**
ER1'	3.1%	18.6%	26.3%	38.7%	242	371
ER1'(MSGen)	**3.6%**	**22.2%**	**28.9%**	**41.2%**	**221**	**335**
GP19	3.1%	9.3%	15.5%	24.2%	253	391
GP19(MSGen)	**3.1%**	**11.9%**	**22.2%**	**27.3%**	**231**	**365**
Ochiai	1.6%	20.1%	24.2%	34.5%	215	363
Ochiai(MSGen)	**2.6%**	**22.2%**	**29.4%**	**38.7%**	**187**	**289**

Top-N, MFR, and MAR The experiments use *Top-N* (*i.e.,* N=1, 5, 10, 20), *MAR*, *MFR* to compare MSGen with the 12 fault localization approaches. Table II presents the distribution among 12 fault localization approaches. As shown in Table II, MSGen achieves higher *Top-N* values and lower *MFR* and *MAR* values compared to all 12 fault localization approaches. It means that MSGen shows promising best localization effectiveness in all scenarios in comparison to the 12 baselines.



(a) *RImp* on fault localization approaches.



(b) *RImp* on subject programs.

Fig. 4. *RImp* comparison of MSGen over the 12 localization approaches.

RImp distribution For a detailed improvement, we adopt *RImp* to evaluate MSGen. Fig. 4 shows the *RImp* scores of using MSGen in two cases: the *RImp* on 12 fault localization approaches in Fig. 4(a) and the *RImp* on 14 subject programs in Fig. 4(b).

As shown in Fig. 4(a), the *RImp* score is less than 100% in all fault localization approaches, meaning that MSGen improves the effectiveness of all the fault localization approaches. Take MLP-FL as an example. The *RImp* score is 67.17%, meaning that MSGen needs to examine 67.17% of the statements that MLP-FL needs to examine for locating all faults of all the 14 subject programs. In other words, MSGen obtains a saving of 33.83% (100%-67.17%=33.83%) over MLP-FL. We can observe that MSGen obtains *RImp* scores ranging from 48.78% in GP03 to 95.68% in Dstar. It means that MSGen gets a maximum saving of 51.22% (100%-48.78%=51.22%) in GP03 and the minimum saving is 4.32% (100%-95.68%=4.32%) in Dstar to locate all faults of the 14 programs.

As shown in Fig. 4(b), the *RImp* score is less than 100% on all programs, meaning that MSGen obtains the improvements on all the programs. The *RImp* score ranges from 56.74% on *math* to 97.19% on *libtiff*. It means that MSGen obtains a maximum saving of 43.26% (100%-56.74%=43.26%) on *math* and a minimum saving of 2.81% (100%-97.19%=2.81%) on *libtiff*.

Overall, MSGen obtains an average saving of 22.68% over the 12 fault localization approaches on all 14 programs, showing that MSGen is effective to improve fault localization.

Statistical comparison To investigate whether the difference between the baselines and MSGen is statistically significant, we adopt Wilcoxon-Signed-Rank Test [49] with a Bonferroni correction [50]. The experiments performed 12 paired Wilcoxon-Signed-Rank tests between MSGen and each of the 12 localization approaches by using *ranks* as the pairs of measurements F(x) and G(y). Each test uses both the 2-tailed and 1-tailed checking at the σ level of 0.05. Specifically, given a localization technique FL1, we use the list of *ranks* of FL1 using MSGen in all faulty versions of all programs as the list of measurements of F(x), while the list of measurements of G(y) is the list of *ranks* of FL1 without MSGen in all faulty versions of all programs. Hence, in the 2-tailed test, FL1 using MSGen has SIMILAR effectiveness as FL1 when H_0 is accepted at the significant level of 0.05. And in the 1-tailed test (right), FL1 using MSGen has WORSE effectiveness than FL1 when H_1 is accepted at the significant level of 0.05. Finally, in the 1-tailed test (left), FL1 using MSGen has BETTER effectiveness than FL1 when H_1 is accepted at the significant level of 0.05.

Table III summarizes the statistical results on this relationship in two cases: the comparison of MSGen over each fault localization approaches in all 14 subject programs and the comparison of MSGen over each program in all 12 fault localization approaches. Take MLP-FL(MSGen) vs MLP-FL and gzip(MSGen) vs gzip as the examples. In case of MLP-FL(MSGen) vs MLP-FL, after applying MSGen to MLP-FL, MSGen obtains 14 BETTER results on 14 out of 14 (14/14=100%) subject programs, 0 (0/14=0%) SIMILAR, and 0(0/14=%) WORSE results. In case of gzip(MSGen) vs gzip,

TABLE III
WILCOXON-SIGNED-RANK TEST OF THE EFFECTIVENESS RELATIONSHIP OF MSGEN OVER 12 FAULT LOCALIZATION APPROACHES.

Comparison on fault localization approaches		Result	Comparison on fault localization approaches		Result
MLP-FL(MSGen) vs MLP-FL	BETTER	14(100%)	CNN-FL(MSGen) vs CNN-FL	BETTER	12(85.7%)
	SIMILAR	0(0%)		SIMILAR	2(14.3%)
	WORSE	0(0%)		WORSE	0(0%)
BiLSTM-FL(MSGen) vs BiLSTM-FL	BETTER	13(92.9%)	DeepFL(MSGen) vs DeepFL	BETTER	11(78.6%)
	SIMILAR	1(7.1%)		SIMILAR	3(21.4%)
	WORSE	0(0%)		WORSE	0(0%)
FLUCCS(MSGen) vs FLUCCS	BETTER	12(85.7%)	ER5(MSGen) vs ER5	BETTER	7(50%)
	SIMILAR	2(14.3%)		SIMILAR	7(50%)
	WORSE	0(0%)		WORSE	0(0%)
GP02(MSGen) vs GP02	BETTER	7(50%)	GP03(MSGen) vs GP03	BETTER	8(57.1%)
	SIMILAR	7(50%)		SIMILAR	6(42.9%)
	WORSE	0(0%)		WORSE	0(0%)
Dstar(MSGen) vs Dstar	BETTER	6(42.9%)	ER1'(MSGen) vs ER1'	BETTER	5(35.7%)
	SIMILAR	8(57.1%)		SIMILAR	9(64.3%)
	WORSE	0(0%)		WORSE	0(0%)
GP19(MSGen) vs GP19	BETTER	8(57.1%)	Ochiai(MSGen) vs Ochiai	BETTER	6(42.9%)
	SIMILAR	6(42.9%)		SIMILAR	8(57.1%)
	WORSE	0(0%)		WORSE	0(0%)
Comparison on subject programs		**Result**	**Comparison on subject programs**		**Result**
gzip(MSGen) vs gzip	BETTER	7(58.3%)	libtiff(MSGen) vs libtiff	BETTER	5(41.7%)
	SIMILAR	5(41.7%)		SIMILAR	7(58.3%)
	WORSE	0(0%)		WORSE	0(0%)
lang(MSGen) vs lang	BETTER	5(41.7%)	closure(MSGen) vs closure	BETTER	6(50%)
	SIMILAR	7(58.3%)		SIMILAR	6(50%)
	WORSE	0(0%)		WORSE	0(0%)
python(MSGen) vs python	BETTER	3(25%)	space(MSGen) vs space	BETTER	9(75%)
	SIMILAR	9(75%)		SIMILAR	3(25%)
	WORSE	0(0%)		WORSE	0(0%)
chart(MSGen) vs chart	BETTER	8(66.7%)	math(MSGen) vs math	BETTER	10(83.3%)
	SIMILAR	4(36.4%)		SIMILAR	2(16.7%)
	WORSE	0(0%)		WORSE	0(0%)
mockito(MSGen) vs mockito	BETTER	11(91.67%)	time(MSGen) vs time	BETTER	7(58.3%)
	SIMILAR	1(8.3%)		SIMILAR	5(41.7%)
	WORSE	0(0%)		WORSE	0(0%)
nanoxml_v1(MSGen) vs nanoxml_v1	BETTER	8(66.7%)	nanoxml_v2(MSGen) vs nanoxml_v2	BETTER	9(75%)
	SIMILAR	4(36.4%)		SIMILAR	3(25%)
	WORSE	0(0%)		WORSE	0(0%)
nanoxml_v3(MSGen) vs nanoxml_v3	BETTER	7(58.3%)	nanoxml_v5(MSGen) vs nanoxml_v5	BETTER	9(75%)
	SIMILAR	5(41.7%)		SIMILAR	3(25%)
	WORSE	0(0%)		WORSE	0(0%)

when locating the faults of the program gzip, MSGen obtains 7 BETTER results on 7 out of 12 (7/12=58.3%) fault localization approaches, 5 SIMILAR results on 5 out of 12 (5/12=41.7%) localization approaches, and 0 (0/0=0%) results.

As shown in Table III, overall, MSGen has 213 BETTER results (213/336= 63.39%), 123 SIMILAR results (123/336=36.61%), and no WORSE result.

Efficiency Based on the existing model-domain failing test samples, MSGen produces new synthesized failing test samples, and we need to evaluate its time cost. Table IV presents the time of producing model-domain failing test samples. As shown in Table IV, MSGen consumes an average of 3.62 seconds, leading to low overhead.

Thus, based on all the results and analysis, we can safely conclude that MSGen significantly improves localization effectiveness, showing that producing synthesized test samples from the information mode domain is potential to improve fault localization.

D. MSGen versus Data Optimization Approaches

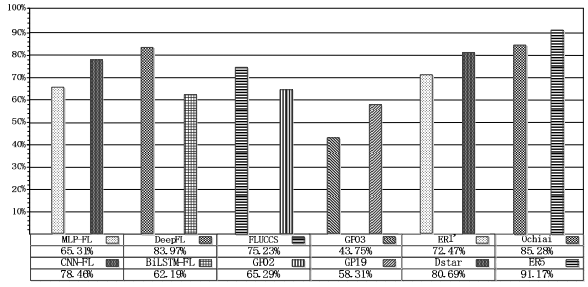
Top-N, MFR, and MAR The experiments use *Top-N* (*i.e.,* N=1, 5, 10, 20), *MAR*, *MFR* to compare MSGen with the two data optimization approaches (*i.e.,* undersampling and resampling). Table V presents the distribution among the two optimization approaches. As shown in Table V, MSGen achieves higher *Top-N* values and lower *MFR* and *MAR* values compared to the two data optimization approaches. It means that MSGen shows promising best localization effectiveness in all scenarios in comparison to the two data optimization approaches.

TABLE IV
TIME COST OF PRODUCING SYNTHESIZED MODEL-DOMAIN FAILING TEST SAMPLES.

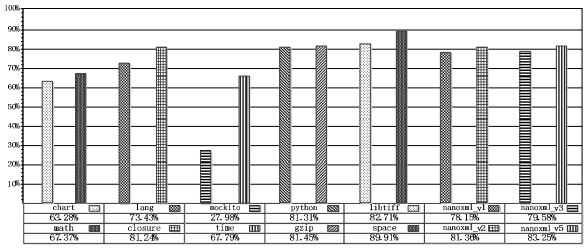
chart	math	lang	closure	mockito	time	python
4.62s	3.13s	3.1s	6.75s	1.52s	5.09s	8.19s
gzip	libtiff	space	nanoxml_v1	nanoxml_v2	nanoxml_v3	nanoxml_v5
8.26s	5.46s	1.17s	0.89s	0.97s	0.74s	0.82s

TABLE V
Top-N, *MAR* AND *MFR* COMPARISON OF MSGEN OVER THE TWO DATA OPTIMIZATION APPROACHES.

Comparison	top-1	top-5	top-10	top-20	MFR	MAR
MLP-FL(undersampling)	0%	1.6%	3.1%	10.8%	225	386
MLP-FL(resampling)	1.6.%	4.1%	9.3%	18.0%	211	344
MLP-FL(MSGen)	**1.6%**	**5.7%**	**11.9%**	**19.1%**	**194**	**337**
CNN-FL(undersampling)	0%	2%	6%	17%	185	273
CNN-FL(resampling)	1.6%	8.2%	15.5%	25.8%	131	257
CNN-FL(MSGen)	**1.6%**	**9.3%**	**16.5%**	**26.3%**	**122**	**248**
BiLSTM-FL(undersampling)	0%	0%	2.1%	9.3%	247	421
BiLSTM-FL(resampling)	0%	3.6%	8.2%	12.4%	215	374
BiLSTM-FL(MSGen)	**0%**	**3.6%**	**9.3%**	**17.0%**	**207**	**369**
DeepFL(undersampling)	0%	3.6%	6.2%	18.6%	202	285
DeepFL(resampling)	1.6%	8.2%	15.5%	25.8%	137	243
DeepFL(MSGen)	**1.6%**	**9.3%**	**17.0%**	**25.8%**	**131**	**239**
FLUCCS(undersampling)	0%	5.4%	11.3%	15.5%	205	339
FLUCCS(resampling)	1.6%	9.3%	15.9%	24.9%	172	297
FLUCCS(MSGen)	**1.6%**	**9.3%**	**17.0%**	**25.8%**	**159**	**288**
ER5(undersampling)	0%	6.2%	8.2%	26.3%	251	439
ER5(resampling)	0%	7.7%	13.5%	25.4%	229	397
ER5(MSGen)	**0%**	**7.7%**	**14.4%**	**27.3%**	**215**	**384**
GP02(undersampling)	1.6%	15.5%	21.8%	38.3%	259	538
GP02(resampling)	2.6%	17.0%	23.2%	39.4%	218	472
GP02(MSGen)	**2.6%**	**17.0%**	**23.2%**	**40.7%**	**221**	**476**
GP03(undersampling)	3.1%	12.4%	16.1%	20.1%	202	347
GP03(resampling)	3.1%	14.4%	17.0%	21.1%	193	331
GP03(MSGen)	**3.1%**	**15.5%**	**18.0%**	**21.1%**	**197**	**326**
Dstar(undersampling)	3.1%	19.1%	24.2%	38.7%	255	361
Dstar(resampling)	3.6%	23.2%	29.4%	43.0%	226	321
Dstar(MSGen)	**3.6%**	**23.2%**	**31.4%**	**46.9%**	**219**	**317**
ER1'(undersampling)	3.1%	18.6%	23.20%	37.1%	247	383
ER1'(resampling)	3.6%	22.2%	27.3%	40.7%	226	343
ER1'(MSGen)	**3.6%**	**22.2%**	**28.9%**	**41.2%**	**221**	**335**
GP19(undersampling)	3.1%	9.3%	14.5%	23.3%	259	404
GP19(resampling)	3.1%	11.9%	21.1%	25.8%	246	375
GP19(MSGen)	**3.1%**	**11.9%**	**22.2%**	**27.3%**	**231**	**365**
Ochiai(undersampling)	1.6%	18.6%	20.6%	34.5%	231	374
Ochiai(resampling)	2.6%	22.1%	29.4%	35.2%	194	327
Ochiai(MSGen)	**2.6%**	**22.2%**	**29.4%**	**38.7%**	**187**	**289**



(a) *RImp* on fault localization approaches.



(b) *RImp* on subject programs.

Fig. 5. *RImp* comparison of MSGen over undersampling.

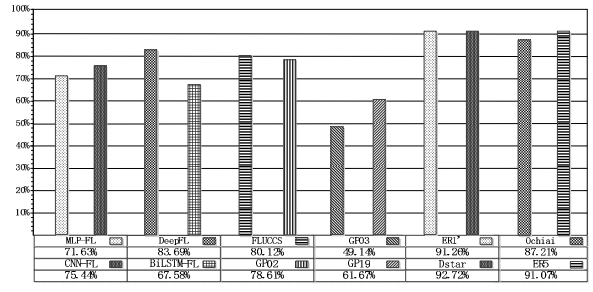
RImp distribution Fig. 5 and Fig. 6 show the *RImp* scores of MSGen over undersampling and resampling respectively. We can observe that the *RImp* scores are less than 100%, meaning that the improvement of using MSGen on fault localization is larger than that of undersampling and resampling. Overall, when using the three data optimization approaches (*i.e.,* MSGen, undersampling, and resampling) for fault localization, MSGen obtains an average saving of 28.16% and 21.47% over undersampling and resampling, respectively. In comparison to

undersampling and resampling, MSGen is significantly more effective to improve fault localization.

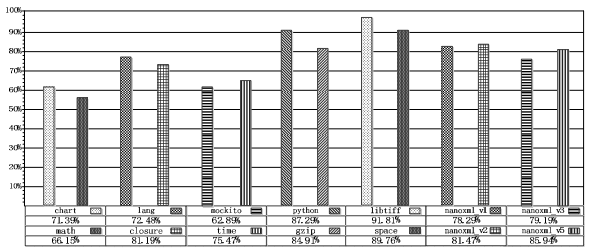
TABLE VI
WILCOXON-SIGNED-RANK TEST OF MSGEN OVER UNDERSAMPLING.

Comparison on fault localization approaches		Result	Comparison on fault localization approaches		Result
MLP-FL(MSGen) vs MLP-FL	BETTER	14(100%)	CNN-FL(MSGen) vs CNN-FL	BETTER	14(100%)
	SIMILAR	0(0%)		SIMILAR	0(0%)
	WORSE	0(0%)		WORSE	0(0%)
BiLSTM-FL(MSGen) vs BiLSTM-FL	BETTER	14(100%)	DeepFL(MSGen) vs DeepFL	BETTER	13(92.9%)
	SIMILAR	0(0%)		SIMILAR	1(7.1%)
	WORSE	0(0%)		WORSE	0(0%)
FLUCCS(MSGen) vs FLUCCS	BETTER	14(100%)	ER5(MSGen) vs ER5	BETTER	7(50%)
	SIMILAR	0(0%)		SIMILAR	7(50%)
	WORSE	0(0%)		WORSE	0(0%)
GP02(MSGen) vs GP02	BETTER	10(71.4%)	GP03(MSGen) vs GP03	BETTER	10(71.4%)
	SIMILAR	2(14.3%)		SIMILAR	4(28.6%)
	WORSE	2(14.3%)		WORSE	0(0%)
Dstar(MSGen) vs Dstar	BETTER	7(50%)	ER1'(MSGen) vs ER1'	BETTER	5(35.7%)
	SIMILAR	5(35.7%)		SIMILAR	7(50%)
	WORSE	2(14.3%)		WORSE	2(14.3%)
GP19(MSGen) vs GP19	BETTER	9(64.3%)	Ochiai(MSGen) vs Ochiai	BETTER	6(42.9%)
	SIMILAR	5(35.7%)		SIMILAR	7(50%)
	WORSE	0(0%)		WORSE	1(7.1%)
Comparison on subject programs		**Result**	**Comparison on subject programs**		**Result**
gzip(MSGen) vs gzip	BETTER	8(66.7%)	libtiff(MSGen) vs libtiff	BETTER	5(41.7%)
	SIMILAR	4(33.3%)		SIMILAR	7(58.3%)
	WORSE	0(0%)		WORSE	0(0%)
lang(MSGen) vs lang	BETTER	9(75%)	closure(MSGen) vs closure	BETTER	8(66.7%)
	SIMILAR	3(25%)		SIMILAR	4(33.3%)
	WORSE	0(0%)		WORSE	0(0%)
python(MSGen) vs python	BETTER	9(75%)	space(MSGen) vs space	BETTER	8(66.7%)
	SIMILAR	3(25%)		SIMILAR	4(33.3%)
	WORSE	0(0%)		WORSE	0(0%)
chart(MSGen) vs chart	BETTER	12(100%)	math(MSGen) vs math	BETTER	9(75%)
	SIMILAR	0(0%)		SIMILAR	2(16.7%)
	WORSE	0(0%)		WORSE	1(8.3%)
mockito(MSGen) vs mockito	BETTER	11(91.7%)	time(MSGen) vs time	BETTER	10(83.3%)
	SIMILAR	1(8.3%)		SIMILAR	2(16.7%)
	WORSE	0(0%)		WORSE	0(0%)
nanoxml_v1(MSGen) vs nanoxml_v1	BETTER	8(66.7%)	nanoxml_v2(MSGen) vs nanoxml_v2	BETTER	8(66.7%)
	SIMILAR	3(25%)		SIMILAR	4(33.3%)
	WORSE	1(8.3%)		WORSE	0(0%)
nanoxml_v3(MSGen) vs nanoxml_v3	BETTER	7(58.3%)	nanoxml_v5(MSGen) vs nanoxml_v5	BETTER	9(75%)
	SIMILAR	5(41.7%)		SIMILAR	3(25%)
	WORSE	0(0%)		WORSE	0(0%)

Statistical comparison Table VI and Table VII summarize the statistical results of MSGen over undersampling and resampling, respectively. Take MLP-FL(MSGen) vs MLP-FL and gzip(MSGen) vs gzip in Talbe VI as the examples. In case of MLP-FL(MSGen) vs MLP-FL, when applying MSGen and undersampling to MLP-FL, MSGen obtain 14 BETTER results over undersampling on 14 out of 14 (14/14=100%) subject programs, 0 (0/14=0%) SIMILAR, and 0(0/14=%) WORSE results. In case of gzip(MSGen) vs gzip, when locating the faults of the program gzip by applying MSGen and undersampling, MSGen obtains 8 BETTER results over undersampling on 8 out of 12 (8/12=66.7%) fault localization



(a) *RImp* on fault localization approaches.



(b) *RImp* on subject programs.
Fig. 6. *RImp* comparison of MSGen over resampling.

TABLE VII
WILCOXON-SIGNED-RANK TEST OF MSGEN OVER RESAMPLING.

Comparison on fault localization approaches		Result	Comparison on fault localization approaches		Result
MLP-FL(MSGen) vs MLP-FL	BETTER	12(85.7%)	CNN-FL(MSGen) vs CNN-FL	BETTER	11(78.6%)
	SIMILAR	2(14.3%)		SIMILAR	2(14.3%)
	WORSE	0(0%)		WORSE	1(7.1%)
BiLSTM-FL(MSGen) vs BiLSTM-FL	BETTER	13(92.9%)	DeepFL(MSGen) vs DeepFL	BETTER	10(71.4%)
	SIMILAR	1(7.1%)		SIMILAR	4(28.6%)
	WORSE	0(0%)		WORSE	0(0%)
FLUCCS(MSGen) vs FLUCCS	BETTER	12(85.7%)	ER5(MSGen) vs ER5	BETTER	5(35.7%)
	SIMILAR	2(14.3%)		SIMILAR	9(64.3%)
	WORSE	0(0%)		WORSE	0(0%)
GP02(MSGen) vs GP02	BETTER	9(64.3%)	GP03(MSGen) vs GP03	BETTER	7(50%)
	SIMILAR	4(28.6%)		SIMILAR	5(35.7%)
	WORSE	1(7.1%)		WORSE	2(14.3%)
Dstar(MSGen) vs Dstar	BETTER	8(57.1%)	ER1'(MSGen) vs ER1'	BETTER	5(35.7%)
	SIMILAR	6(42.9%)		SIMILAR	9(64.3%)
	WORSE	0(0%)		WORSE	0(0%)
GP19(MSGen) vs GP19	BETTER	9(64.3%)	Ochiai(MSGen) vs Ochiai	BETTER	4(28.6%)
	SIMILAR	5(35.7%)		SIMILAR	10(71.4%)
	WORSE	0(0%)		WORSE	0(0%)
Comparison on subject programs		**Result**	**Comparison on subject programs**		**Result**
gzip(MSGen) vs gzip	BETTER	7(58.3%)	libtiff(MSGen) vs libtiff	BETTER	5(41.7%)
	SIMILAR	5(41.7%)		SIMILAR	7(58.3%)
	WORSE	0(0%)		WORSE	0(0%)
lang(MSGen) vs lang	BETTER	5(41.7%)	closure(MSGen) vs closure	BETTER	6(50%)
	SIMILAR	7(58.3%)		SIMILAR	6(50%)
	WORSE	0(0%)		WORSE	0(0%)
python(MSGen) vs python	BETTER	3(25%)	space(MSGen) vs space	BETTER	9(75%)
	SIMILAR	9(75%)		SIMILAR	3(25%)
	WORSE	0(0%)		WORSE	0(0%)
chart(MSGen) vs chart	BETTER	8(66.7%)	math(MSGen) vs math	BETTER	10(83.3%)
	SIMILAR	4(33.3%)		SIMILAR	2(16.7%)
	WORSE	0(0%)		WORSE	0(0%)
mockito(MSGen) vs mockito	BETTER	11(91.7%)	time(MSGen) vs time	BETTER	7(58.3%)
	SIMILAR	1(8.3%)		SIMILAR	5(41.7%)
	WORSE	0(0%)		WORSE	0(0%)
nanoxml_v1(MSGen) vs nanoxml_v1	BETTER	8(66.7%)	nanoxml_v2(MSGen) vs nanoxml_v2	BETTER	9(75%)
	SIMILAR	4(33.3%)		SIMILAR	3(25%)
	WORSE	0(0%)		WORSE	0(0%)
nanoxml_v3(MSGen) vs nanoxml_v3	BETTER	6(50%)	nanoxml_v5(MSGen) vs nanoxml_v5	BETTER	9(75%)
	SIMILAR	6(50%)		SIMILAR	3(25%)
	WORSE	0(0%)		WORSE	0(0%)

approaches, 4 SIMILAR results on 4 out of 12 (4/12=33.3%) localization approaches, and 0 (0/12=0%) results.

Overall, compared with undersampling, MSGen has 244 BETTER results (244/336= 72.62%), 83 SIMILAR results (83/336=24.70%), and 9 WORSE results (9/308=2.68%); compared with resampling, MSGen has 208 BETTER results (208/336= 61.91%), 124 SIMILAR results (124/336=36.90%), and 4 WORSE results (4/336=1.19%).

V. DISCUSSION

A. Why MSGen Is Effective?

TABLE VIII
WILCOXON-SIGNED-RANK TEST OF MSGEN OVER MSGEN WITHOUT PRESERVING THE COMMON FEATURES.

Comparison on fault localization approaches		Result	Comparison on fault localization approaches		Result
MLP-FL(MSGen) vs MLP-FL	BETTER	14(100%)	CNN-FL(MSGen) vs CNN-FL	BETTER	14(100%)
	SIMILAR	0(0%)		SIMILAR	0(0%)
	WORSE	0(0%)		WORSE	0(0%)
BiLSTM-FL(MSGen) vs BiLSTM-FL	BETTER	14(100%)	DeepFL(MSGen) vs DeepFL	BETTER	14(100%)
	SIMILAR	0(0%)		SIMILAR	0(0%)
	WORSE	0(0%)		WORSE	0(0%)
FLUCCS(MSGen) vs FLUCCS	BETTER	14(100%)	ER5(MSGen) vs ER5	BETTER	14(100%)
	SIMILAR	0(0%)		SIMILAR	0(0%)
	WORSE	0(0%)		WORSE	0(0%)
GP02(MSGen) vs GP02	BETTER	14(100%)	GP03(MSGen) vs GP03	BETTER	14(100%)
	SIMILAR	0(0%)		SIMILAR	0(0%)
	WORSE	0(0%)		WORSE	0(0%)
Dstar(MSGen) vs Dstar	BETTER	14(100%)	ER1'(MSGen) vs ER1'	BETTER	14(100%)
	SIMILAR	0(0%)		SIMILAR	0(0%)
	WORSE	0(0%)		WORSE	0(0%)
GP19(MSGen) vs GP19	BETTER	14(100%)	Ochiai(MSGen) vs Ochiai	BETTER	14(100%)
	SIMILAR	0(0%)		SIMILAR	0(0%)
	WORSE	0(0%)		WORSE	0(0%)
Comparison on subject programs		**Result**	**Comparison on subject programs**		**Result**
gzip(MSGen) vs gzip	BETTER	12(100%)	libtiff(MSGen) vs libtiff	BETTER	12(100%)
	SIMILAR	0(0%)		SIMILAR	0(0%)
	WORSE	0(0%)		WORSE	0(0%)
lang(MSGen) vs lang	BETTER	12(100%)	closure(MSGen) vs closure	BETTER	12(100%)
	SIMILAR	0(0%)		SIMILAR	0(0%)
	WORSE	0(0%)		WORSE	0(0%)
python(MSGen) vs python	BETTER	12(100%)	space(MSGen) vs space	BETTER	12(100%)
	SIMILAR	0(0%)		SIMILAR	0(0%)
	WORSE	0(0%)		WORSE	0(0%)
chart(MSGen) vs chart	BETTER	12(100%)	math(MSGen) vs math	BETTER	12(100%)
	SIMILAR	0(0%)		SIMILAR	0(0%)
	WORSE	0(0%)		WORSE	0(0%)
mockito(MSGen) vs mockito	BETTER	12(100%)	time(MSGen) vs time	BETTER	12(100%)
	SIMILAR	0(0%)		SIMILAR	0(0%)
	WORSE	0(0%)		WORSE	0(0%)
nanoxml_v1(MSGen) vs nanoxml_v1	BETTER	12(100%)	nanoxml_v2(MSGen) vs nanoxml_v2	BETTER	12(100%)
	SIMILAR	0(0%)		SIMILAR	0(0%)
	WORSE	0(0%)		WORSE	0(0%)
nanoxml_v3(MSGen) vs nanoxml_v3	BETTER	12(100%)	nanoxml_v5(MSGen) vs nanoxml_v5	BETTER	12(100%)
	SIMILAR	0(0%)		SIMILAR	0(0%)
	WORSE	0(0%)		WORSE	0(0%)

The prior work [30] has shown the data covering the common features of all failing tests (*i.e.,* minimum suspicious set) are beneficial (or safe) for fault localization, where a minimum suspicious set is defined as those statements executed by all failing tests. SBFL and DLFL share a similar idea that when a statement occurs in a failing test, its suspiciousness

increases; while the statement occurs in a passing test, its suspiciousness decreases. Their difference is that DLFL uses neural networks while SBFL utilizes statistical correlation coefficients to implement a similar idea. It is intuitive that a faulty statement should be executed to cause a failure and the faulty statement should generally be in the minimum suspicious set except some cases like multiple faults. If a synthesized model-domain failing test sample covers the minimum suspicious set, it usually covers the faulty statement, and thus the suspiciousness of the faulty statement should increase. Thus, MSGen uses the minimum variability of minority class feature space to produce synthesized model-domain failing test samples, *i.e.,* it preserves the common features of all model-domain failing test samples, namely the minimum suspicious set. If we use MSGen without preserving the common features (*i.e.,* without covering the minimum suspicious set), MSGen should significantly outperform the one without preserving the common features.

Table VIII shows the statistical results of the comparison between MSGen and MSGen without preserving the common features. The statistical comparison uses the Wilcoxon-Signed-Rank test at the σ level of 0.05. As shown in Table VIII, MSGen obtains all BETTER results over the one without preserving the common features in all 12 fault localization approaches and 14 subject programs, indicating that preserving the common features is the key factor of why MSGen is effective.

B. Effect of Passing Tests on FL Effectiveness

It is natural to raise a question on why MSGen considers the distance (*i.e.,* Euclidean distance [45]) of the existing model-domain failing test samples only. If we produce those synthesized samples by maximizing the distance from the existing model-domain passing test samples, it is reasonable to fix these synthesized samples as a failing label since they are far from the existing passing samples. Therefore, we compare MSGen with the two approaches using Euclidean distance of passing tests. One approach (denoted as MaxP) produces synthesized model-domain failing test samples by maximizing the distance from the existing model-domain passing test samples; the other one (denoted as MinF+MaxP) produces synthesized model-domain failing test samples by minimizing the distance from the existing model-domain failing test samples and meanwhile maximizing the distance from the existing model-domain passing test samples.

Tables IX and X show the statistical results of the comparison of MSGen over MaxP and MinF+MaxP respectively. The statistical comparison uses the Wilcoxon-Signed-Rank test at the σ level of 0.05. As shown in the tables, MSGen obtains BETTER results in most fault localization approaches and these subject programs, and no WORSE results, indicating that MSGen is more effective than MaxP and MinF+MaxP. For failing tests, their executions can always include the faulty statement. In contrast, for passing tests, their executions cannot be guaranteed to be free of a faulty statement, leading to a coincidental correctness problem [30]. If we consider the distance of passing tests, the coincidental correctness problem can cause new samples to be far away from the faulty statement. This is why our approach significantly outperforms MaxP and MinF+MaxP, and does not consider passing tests to avoid the threat caused by the coincidental correctness problem.

TABLE IX
WILCOXON-SIGNED-RANK TEST OF MSGEN OVER MAXP.

Comparison on fault localization approaches		Result	Comparison on fault localization approaches		Result
MLP-FL(MSGen) vs MLP-FL	BETTER	14(100%)	CNN-FL(MSGen) vs CNN-FL	BETTER	14(100%)
	SIMILAR	0(0%)		SIMILAR	0(0%)
	WORSE	0(0%)		WORSE	0(0%)
BiLSTM-FL(MSGen) vs BiLSTM-FL	BETTER	14(100%)	DeepFL(MSGen) vs DeepFL	BETTER	13(92.9%)
	SIMILAR	0(0%)		SIMILAR	1(7.1%)
	WORSE	0(0%)		WORSE	0(0%)
FLUCCS(MSGen) vs CNN-FL	BETTER	14(100%)	ER5(MSGen) vs ER5	BETTER	10(71.4%)
	SIMILAR	0(0%)		SIMILAR	4(28.6%)
	WORSE	0(0%)		WORSE	0(0%)
GP02(MSGen) vs GP02	BETTER	13(92.9%)	GP03(MSGen) vs GP03	BETTER	10(71.4%)
	SIMILAR	1(7.1%)		SIMILAR	4(28.6%)
	WORSE	0(0%)		WORSE	0(0%)
Dstar(MSGen) vs Dstar	BETTER	9(64.3%)	ER1'(MSGen) vs ER1'	BETTER	7(50%)
	SIMILAR	5(35.7%)		SIMILAR	7(50%)
	WORSE	0(0%)		WORSE	0(0%)
GP19(MSGen) vs GP19	BETTER	10(71.4%)	Ochiai(MSGen) vs Ochiai	BETTER	8(42.9%)
	SIMILAR	4(28.6%)		SIMILAR	6(50%)
	WORSE	0(0%)		WORSE	0(0%)
Comparison on subject programs		**Result**	**Comparison on subject programs**		**Result**
gzip(MSGen) vs gzip	BETTER	9(75%)	libtiff(MSGen) vs libtiff	BETTER	9(75%)
	SIMILAR	3(25%)		SIMILAR	3(25%)
	WORSE	0(0%)		WORSE	0(0%)
lang(MSGen) vs lang	BETTER	11(91.7%)	closure(MSGen) vs closure	BETTER	9(75%)
	SIMILAR	1(8.3%)		SIMILAR	3(25%)
	WORSE	0(0%)		WORSE	0(0%)
python(MSGen) vs python	BETTER	11(91.7%)	space(MSGen) vs space	BETTER	11(91.7%)
	SIMILAR	1(8.3%)		SIMILAR	1(8.3%)
	WORSE	0(0%)		WORSE	0(0%)
chart(MSGen) vs chart	BETTER	12(100%)	math(MSGen) vs math	BETTER	11(91.7%)
	SIMILAR	0(0%)		SIMILAR	1(8.3%)
	WORSE	0(0%)		WORSE	0(0%)
mockito(MSGen) vs mockito	BETTER	12(100%)	time(MSGen) vs time	BETTER	9(89.8%)
	SIMILAR	0(0%)		SIMILAR	2(18.2%)
	WORSE	0(0%)		WORSE	0(0%)
nanoxml_v1(MSGen) vs nanoxml_v1	BETTER	9(75%)	nanoxml_v2(MSGen) vs nanoxml_v2	BETTER	11(91.7%)
	SIMILAR	3(25%)		SIMILAR	1(8.3%)
	WORSE	0(0%)		WORSE	0(0%)
nanoxml_v3(MSGen) vs nanoxml_v3	BETTER	9(75%)	nanoxml_v5(MSGen) vs nanoxml_v5	BETTER	11(91.7%)
	SIMILAR	3(25%)		SIMILAR	1(8.3%)
	WORSE	0(0%)		WORSE	0(0%)

TABLE X
WILCOXON-SIGNED-RANK TEST OF MSGEN OVER MINF+MAXP.

Comparison on fault localization approaches		Result	Comparison on fault localization approaches		Result
MLP-FL(MSGen) vs MLP-FL	BETTER	13(92.9%)	CNN-FL(MSGen) vs CNN-FL	BETTER	11(78.6%)
	SIMILAR	1(7.1%)		SIMILAR	3(21.4%)
	WORSE	0(0%)		WORSE	0(0%)
BiLSTM-FL(MSGen) vs BiLSTM-FL	BETTER	14(100%)	DeepFL(MSGen) vs DeepFL	BETTER	10(71.4%)
	SIMILAR	0(0%)		SIMILAR	4(28.6%)
	WORSE	0(0%)		WORSE	0(0%)
FLUCCS(MSGen) vs FLUCCS	BETTER	11(78.6%)	ER5(MSGen) vs ER5	BETTER	9(64.3%)
	SIMILAR	3(21.4%)		SIMILAR	5(35.7%)
	WORSE	0(0%)		WORSE	0(0%)
GP02(MSGen) vs GP02	BETTER	9(64.3%)	GP03(MSGen) vs GP03	BETTER	8(57.1%)
	SIMILAR	5(35.7%)		SIMILAR	6(42.9%)
	WORSE	0(0%)		WORSE	0(0%)
Dstar(MSGen) vs Dstar	BETTER	8(57.1%)	ER1'(MSGen) vs ER1'	BETTER	8(57.1%)
	SIMILAR	6(42.9%)		SIMILAR	6(42.9%)
	WORSE	0(0%)		WORSE	0(0%)
GP19(MSGen) vs GP19	BETTER	11(78.6%)	Ochiai(MSGen) vs Ochiai	BETTER	7(50%)
	SIMILAR	3(21.4%)		SIMILAR	7(50%)
	WORSE	0(0%)		WORSE	0(0%)
Comparison on subject programs		**Result**	**Comparison on subject programs**		**Result**
gzip(MSGen) vs gzip	BETTER	9(75%)	libtiff(MSGen) vs libtiff	BETTER	8(66.7%)
	SIMILAR	3(25%)		SIMILAR	4(33.3%)
	WORSE	0(0%)		WORSE	0(0%)
lang(MSGen) vs lang	BETTER	7(58.3%)	closure(MSGen) vs closure	BETTER	8(66.7%)
	SIMILAR	5(41.7%)		SIMILAR	4(33.3%)
	WORSE	0(0%)		WORSE	0(0%)
python(MSGen) vs python	BETTER	6(50%)	space(MSGen) vs space	BETTER	11(91.7%)
	SIMILAR	6(50%)		SIMILAR	1(8.3%)
	WORSE	0(0%)		WORSE	0(0%)
chart(MSGen) vs chart	BETTER	9(75%)	math(MSGen) vs math	BETTER	11(91.7%)
	SIMILAR	3(25%)		SIMILAR	1(8.3%)
	WORSE	0(0%)		WORSE	0(0%)
mockito(MSGen) vs mockito	BETTER	12(100%)	time(MSGen) vs time	BETTER	9(75%)
	SIMILAR	0(0%)		SIMILAR	3(25%)
	WORSE	0(0%)		WORSE	0(0%)
nanoxml_v1(MSGen) vs nanoxml_v1	BETTER	9(75%)	nanoxml_v2(MSGen) vs nanoxml_v2	BETTER	11(91.7%)
	SIMILAR	3(25%)		SIMILAR	1(8.3%)
	WORSE	0(0%)		WORSE	0(0%)
nanoxml_v3(MSGen) vs nanoxml_v3	BETTER	8(66.7%)	nanoxml_v5(MSGen) vs nanoxml_v5	BETTER	11(91.7%)
	SIMILAR	4(33.3%)		SIMILAR	1(8.3%)
	WORSE	0(0%)		WORSE	0(0%)

C. Effect of Balanced Tests on FL Effectiveness

Many studies have found that a class-balanced test suite is useful for fault localization [12], [21]. We use MSGen to generate different ratios of model-level failing tests to verify whether balanced tests are better than unbalanced ones for fault localization. Specifically, we use the ratio $\theta = Pnum/Fnum$, where *Pnum* and *Fnum* denote the number of passing tests and the number of failing tests. We generate different test suites with $\theta = 0.5$, 1 and 1.5. Table XI shows the statistical results of MSGen with different ratios. The results show that MSGen obtains most BETTER results with balanced tests, *i.e.,* balanced tests are better than unbalanced ones for fault localization.

TABLE XI
STATISTICAL RESULTS ON MSGEN USING DIFFERENT RATIOS.

Comparison on fault localization approaches					
$(\theta = 1)$ vs $(\theta = 1.5)$		Result	$(\theta = 1)$ vs $(\theta = 0.5)$		Result
MLP-FL	BETTER	10(71.4%)	MLP-FL	BETTER	11(78.6%)
	SIMILAR	2(14.3%)		SIMILAR	0(0%)
	WORSE	2(14.3%)		WORSE	3(21.4%)
CNN-FL	BETTER	9(64.3%)	CNN-FL	BETTER	10(71.4%)
	SIMILAR	2(14.3%)		SIMILAR	2(14.3%)
	WORSE	3(21.4%)		WORSE	2(14.3%)
BiLSTM-FL	BETTER	9(64.3%)	BiLSTM-FL	BETTER	10(71.4%)
	SIMILAR	3(21.4%)		SIMILAR	1(7.1%)
	WORSE	2(14.3%)		WORSE	3(21.4%)
DeepFL	BETTER	11(78.6%)	DeepFL	BETTER	10(71.4%)
	SIMILAR	2(14.3%)		SIMILAR	2(14.3%)
	WORSE	1(7.1%)		WORSE	2(14.3%)
FLUCCS	BETTER	10(71.4%)	FLUCCS	BETTER	11(78.6%)
	SIMILAR	2(14.3%)		SIMILAR	1(7.1%)
	WORSE	2(14.3%)		WORSE	2(14.3%)
ER5	BETTER	9(64.3%)	ER5	BETTER	9(64.3%)
	SIMILAR	1(7.1%)		SIMILAR	3(21.4%)
	WORSE	4(28.6%)		WORSE	2(14.3%)
GP02	BETTER	7(50%)	GP02	BETTER	9(64.3%)
	SIMILAR	4(28.6%)		SIMILAR	1(7.1%)
	WORSE	3(21.4%)		WORSE	4(28.6%)
GP03	BETTER	10(71.4%)	GP03	BETTER	9(64.3%)
	SIMILAR	0(0%)		SIMILAR	2(14.3%)
	WORSE	4(28.6%)		WORSE	3(21.4%)
Dstar	BETTER	9(64.3%)	Dstar	BETTER	10(71.4%)
	SIMILAR	2(14.3%)		SIMILAR	2(14.3%)
	WORSE	3(21.4%)		WORSE	2(14.3%)
ER1'	BETTER	11(78.6%)	ER1'	BETTER	9(64.3%)
	SIMILAR	2(14.3%)		SIMILAR	3(21.4%)
	WORSE	1(7.1%)		WORSE	2(14.3%)
GP19	BETTER	10(71.4%)	GP19	BETTER	11(78.6%)
	SIMILAR	3(21.4%)		SIMILAR	2(14.3%)
	WORSE	1(7.1%)		WORSE	1(7.1%)
Ochiai	BETTER	7(50%)	BiLSTM-FL	BETTER	8(57.1%)
	SIMILAR	4(28.6%)		SIMILAR	2(14.3%)
	WORSE	3(21.4%)		WORSE	4(28.6%)

D. Threats to Validity

Our experiments use deep learning-based fault localization approaches, meaning that the fault localization results are not the same given different training times. That drawback is caused by the characteristic of neural networks. To make the results more reliable, we followed the convention by repeating the fault localization process, *i.e.,* we computed ten times and used the average score as the results for the experimental study.

Our approach uses the minimum suspicious set which is derived from the existing state-of-the-art localization approaches. It may not hold in some cases where these localization approaches suffer from, *e.g.,* multiple faults. This limitation of the existing localization approaches will affect our approach. We can apply the clustering technology (*e.g.,* [51]) to alleviate the problem via transforming the context of multiple faults into that of single faults.

Another threat is the subject programs. Although we adopt the widely used subject programs, there are still many unknown and complicated factors in real debugging that could affect the experiment results. Thus, it is worthwhile to use more large-sized programs to further strengthen the experimental results.

VI. RELATED WORK

In the field of machine learning and fault localization, the class proportion of datasets has been widely studied. Wong *et al.* [52] show that the class imbalance phenomenon of datasets has an influence on the efficacy of classification. Japkowizc and Shaju [53] empirically find that the class imbalance phenomenon of the training set causes a negative impact to classification problems. Baudry *et al.* [28] conduct the experiments and show that fewer test cases could achieve the same fault localization effectiveness. Hao *et al.* [29] improve the localization effectiveness by using test suite reduction techniques. However, Yu *et al.* [54] suggest that test suite reduction techniques may reduce the effectiveness of fault localization. Gong *et al.* [20] conduct an experimental study showing that the identical number of passing test cases and the failing test cases is beneficial for fault localization. These test generation approaches for fault localization rarely generate failing tests, but instead optimize or generate passing tests for fault localization. In contrast, our approach is to produce failing test samples from the perspective of data synthesis, rather than to optimize or generate passing tests for fault localization. Furthermore, our approach is easier to produce synthesized failing test samples from the model domain and requires no execution of the synthesized data to obtain the result label.

There are also several pieces of work focusing on failure reproduction. Jin *et al.* [23] propose BugRedux, which collects different types of execution data in the field and mimics the observed field failures for F3 technique [22]. Soltani *et al.* [24], [25] use search-based software testing for crash reproduction. Bohme *et al.* [26] introduce a semi-automatic repair technique LEARN2FIX, the first human-in-the-loop based on the user who is reporting a bug is available. Gabin *et al.* [27] propose QFID, which augments the failing test cases with automatically generated test data and elicit oracles from a human developer to label the test cases. Although the reproducing failure are real failing ones, they are difficult to generate enough number of test cases to solve the imbalance problem for fault localization due to the large computing cost and high complexity (*e.g.,* symbolic execution and manual inspection). Unlike these approaches, we seek a different perspective and a simple way to address the class imbalance problem from the model-domain rather than the input-domain.

VII. CONCLUSION AND FUTURE WORK

In this paper, we propose MSGen which generates synthesized test samples from the model domain, rather than generating real tests from the input domain, to improve fault localization. MSGen identifies the nearest neighbors from the existing model-domain failing test samples, and computes the difference between each failing test and its one nearest neighbor to generate new synthesized model-domain test samples. The experimental results on 12 state-of-the-art fault localization and two data optimization approaches show that MSGen can significantly improve fault localization effectiveness with up to 51.22%.

In future, we plan to extend our approach MSGen to the multiple-fault scenario. We also plan to explore more on model domain for synthesized test generation.

ACKNOWLEDGMENTS

This work is supported by the National Key Research and Development Project of China (No. 2020YFB1711900), the National Natural Science Foundation of China (No. 62002034), the Fundamental Research Funds for the Central Universities (No. 2022CDJKYJH001) and the Natural Science Foundation of Chongqing (No. cstc2021jcyj-msxmX0538).

REFERENCES

[1] W. E. Wong, R. Gao, Y. Li, A. Rui, and F. Wotawa, "A survey on software fault localization," *IEEE Transactions on Software Engineering(TSE)*, vol. 42, no. 8, pp. 707–740, 2016.

[2] C. Parnin and A. Orso, "Are automated debugging techniques actually helping programmers?" in *International Symposium on Software Testing and Analysis(ISSTA 2011)*, 2011, pp. 199–209.

[3] Y. Guo, N. Li, J. Offutt, and A. Motro, "Exoneration-based fault localization for sql predicates," *Journal of Systems and Software*, vol. 147, pp. 230–245, 2019.

[4] T. D. B. Le, R. J. Oentaryo, and D. Lo, "Information retrieval and spectrum based bug localization: better together," in *Joint Meeting on Foundations of Software Engineering(FSE 2015)*, 2015, pp. 579–590.

[5] C. Sun and S. C. Khoo, "Mining succinct predicated bug signatures," in *Joint Meeting on Foundations of Software Engineering(FSE 2013)*, 2013, pp. 576–586.

[6] J. Kim, J. Kim, and E. Lee, "Vfl: Variable-based fault localization," *Information and software technology*, 2019.

[7] S. Moon, Y. Kim, M. Kim, and S. Yoo, "Ask the mutants: Mutating faulty programs for fault localization," in *IEEE Seventh International Conference on Software Testing, Verification and Validation(ICST 2014)*, 2014, pp. 153–162.

[8] M. Papadakis and Y. L. Traon, *Metallaxis-FL: Mutation-based fault localization*. John Wiley and Sons Ltd., 2015.

[9] X. Xie, T. Y. Chen, F. C. Kuo, and B. Xu, "A theoretical analysis of the risk evaluation formulas for spectrum-based fault localization," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 22, no. 4, p. 31, 2013.

[10] L. Naish and Hua, "A model for spectra-based software diagnosis," *Acm Transactions on Software Engineering and Methodology*, vol. 20, no. 3, pp. 1–32, 2011.

[11] Z. Zhang, Y. Lei, X. Mao, and P. Li, "CNN-FL: An effective approach for localizing faults using convolutional neural networks," in *the 26th International Conference on Software Analysis, Evolution and Reengineering (SANER 2019)*. IEEE, 2019, pp. 445–455.

[12] Z. Zhang, Y. Lei, X. Mao, M. Yan, L. Xu, and J. Wen, "Improving deep-learning-based fault localization with resampling," *Journal of Software: Evolution and Process*, vol. 33, no. 3, p. e2312, 2021.

[13] X. Li, W. Li, Y. Zhang, and L. Zhang, "DeepFL: Integrating multiple fault diagnosis dimensions for deep fault localization," in *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 2019)*, 2019, pp. 169–180.

[14] J. Sohn and S. Yoo, "Fluccs: Using code and change metrics to improve fault localization," in *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 2017)*, 2017, pp. 273–283.

[15] Z. Zhang, Y. Lei, X. Mao, M. Yan, L. Xu, and X. Zhang, "A study of effectiveness of deep learning in locating real faults," *Information and Software Technology*, vol. 131, p. 106486, 2021.

[16] S. Pearson, J. Campos, and Just, "Evaluating and improving fault localization," in *International Conference on Software Engineering*, 2017.

[17] H. J. Lee, L. Naish, and K. Ramamohanarao, "Effective software bug localization using spectral frequency weighting function," in *Proceedings of the 34th Annual Computer Software and Applications Conference (COMPSAC 2010),*. IEEE, 2010, pp. 218–227.

[18] Y. Lei, X. Mao, M. Zhang, J. Ren, and Y. Jiang, "Toward understanding information models of fault localization: Elaborate is not always better," in *The 41st Annual Computer Software and Applications Conference(COMPSAC 2017)*, 2017, pp. 57–66.

[19] W. Zheng, D. Hu, and J. Wang, "Fault localization analysis based on deep neural network," *Mathematical Problems in Engineering,2016*, vol. 2016, pp. 1–11, 2016.

[20] C. Gong, Z. Zheng, W. Li, and P. Hao, "Effects of class imbalance in test suites: An empirical study of spectrum-based fault localization," in *Proceedings of the 36th Annual Computer Software and Applications Conference Workshops*, 2012, pp. 470–475.

[21] L. Zhang, L. Yan, Z. Zhang, J. Zhang, W. Chan, and Z. Zheng, "A theoretical analysis on cloning the failed test cases to improve spectrum-based fault localization," *Journal of Systems and Software*, vol. 129, pp. 35–57, 2017.

[22] W. Jin and A. Orso, "F3: Fault localization for field failures," in *Proceedings of the 2013 International Symposium on Software Testing and Analysis*, ser. ISSTA 2013. New York, NY, USA: ACM, 2013, pp. 213–223.

[23] ——, "Bugredux: Reproducing field failures for in-house debugging," in *Proceedings of the 34th International Conference on Software Engineering (ICSE)*, 2012.

[24] M. Soltani, P. Derakhshanfar, A. Panichella, X. Devroey, A. Zaidman, and A. van Deursen, "Single-objective versus multi-objectivized optimization for evolutionary crash reproduction," in *Search-Based Software Engineering*, T. E. Colanzi and P. McMinn, Eds. Cham: Springer International Publishing, 2018, pp. 325–340.

[25] M. Soltani, P. Derakhshanfar, X. Devroey, and A. van Deursen, "A benchmark-based evaluation of search-based crash reproduction," *Empirical Software Engineering*, vol. 25, no. 1, pp. 96–138, Jan 2020.

[26] M. Böhme, C. Geethal, and V.-T. Pham, "Human-in-the-loop automatic program repair," in *2020 IEEE 13th International Conference on Software Testing, Validation and Verification (ICST)*, 2020, pp. 274–285.

[27] G. An and S. Yoo, "Human-in-the-loop fault localisation using efficient test prioritisation of generated tests," *CoRR*, vol. abs/2104.06641, 2021.

[28] B. Baudry, "Improving test suites for efficient fault localization," in *International Conference on Software Engineering(ICSE 2006)*, 2006, pp. 82–91.

[29] D. Hao, Y. Pan, L. Zhang, W. Zhao, H. Mei, and J. Sun, "A similarity-aware approach to testing based fault localization," in *Ieee International Conference on Automated Software Engineering (ASE 2005)*, 2005, pp. 291–294.

[30] Y. Lei, C. Sun, X. Mao, and Z. Su, "How test suites impact fault localization starting from the size," *IET Software*, vol. 12, no. 3, pp. 190–205, 2018.

[31] H. He and E. A. Garcia, "Learning from imbalanced data," *IEEE Transactions on Knowledge and Data Engineering(TKDE)*, no. 9, pp. 1263–1284, 2008.

[32] B. Krawczyk, "Learning from imbalanced data: open challenges and future directions," *Progress in Artificial Intelligence*, vol. 5, no. 4, pp. 221–232, 2016.

[33] C. Shorten and T. M. Khoshgoftaar, "A survey on image data augmentation for deep learning," *Journal of Big Data*, vol. 6, no. 1, pp. 1–48, 2019.

[34] Y. Xian, T. Lorenz, B. Schiele, and Z. Akata, "Feature generating networks for zero-shot learning," in *Proceedings of the IEEE conference on Computer Vision and Pattern Recognition (CVPR)*, 2018, pp. 5542–5551.

[35] Y. Xian, S. Sharma, B. Schiele, and Z. Akata, "f-vaegan-d2: A feature generating framework for any-shot learning," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2019, pp. 10 275–10 284.

[36] F. Zhou, S. Huang, and Y. Xing, "Deep semantic dictionary learning for multi-label image classification," *The 35th AAAI Conference on Artificial Intelligence (AAAI)*, 2021.

[37] C. Tantithamthavorn, A. E. Hassan, and K. Matsumoto, "The Impact of Class Rebalancing Techniques on the Performance and Interpretation of Defect Prediction Models," *IEEE Transactions on Software Engineering*, pp. 1–22, 2018.

[38] H. Wang, B. Du, J. He, Y. Liu, and X. Chen, "Ietcr: An information entropy based test case reduction strategy for mutation-based fault localization," *IEEE Access*, vol. 8, pp. 124 297–124 310, 2020.

[39] K. Jarrett, K. Kavukcuoglu, M. Ranzato, and Y. Lecun, "What is the best multi-stage architecture for object recognition?" in *IEEE International Conference on Computer Vision(ICCV 2010)*, 2010, pp. 2146 – 2153.

[40] Y. Lecun, F. J. Huang, and Bottou, "Learning methods for generic object recognition with invariance to pose and lighting," in *Proceedings of the 2004 IEEE Computer Society Conference on Computer Vision and Pattern Recognition(CVPR 2014)*, 2004, pp. II–97–104 Vol.2.

[41] R. R. H. Lee, R. Grosse and A. Ng, "Convolutional deep belief networks for scalable unsupervised learning of hierarchical representations," in *Proceedings of the 26th Annual International Conference on Machine Learning(ICML 2009)*, 2009, pp. 609–616.

[42] J. D. N. Pinto, D. Doukhan and D. Cox, "A high-throughput screening approach to discovering good forms of biologically inspired visual representation," in *PLoS computational biology*, 2009, p. vol.5.

[43] S. C. Turaga, J. F. Murray, V. Jain, F. Roth, M. Helmstaedter, K. Briggman, W. Denk, and H. S. Seung, "Convolutional networks can learn to

generate affinity graphs for image segmentation," *Neural Computation*, vol. 22, no. 2, pp. 511–538, 2010.

[44] X. Xie, F. C. Kuo, T. Y. Chen, S. Yoo, and M. Harman, *Provably Optimal and Human-Competitive Results in SBSE for Spectrum Based Fault Localisation*. Springer Berlin Heidelberg, 2013.

[45] N. V. Chawla, K. W. Bowyer, L. O. Hall, and W. P. Kegelmeyer, "Smote: Synthetic minority over-sampling technique," 2011.

[46] V. Debroy, W. E. Wong, X. Xu, and B. Choi, "A grouping-based strategy to improve the effectiveness of fault localization techniques," in *International Conference on Quality Software(QSIC 2010)*, 2010, pp. 13–22.

[47] L. C. Briand, Y. Labiche, and X. Liu, "Using machine learning to support debugging with tarantula," in *The IEEE International Symposium on Software Reliability(ISSRE 2007)*, 2007, pp. 137–146.

[48] Y. Lei, X. Mao, Z. Dai, and C. Wang, "Effective statistical fault localization using program slices," in *Computer Software and Applications Conference(COMPSAC 2012)*, 2012, pp. 1–10.

[49] G. W. Corder and D. I. Foreman, *Nonparametric Statistics for Non-Statisticians: A Step-by-Step Approach*. International Statistical Review, 2010, vol. 78.

[50] H. Abdi, "The bonferonni and Šidák corrections for multiple comparisons," *Encyclopedia of measurement and statistics*, vol. 3, p. 103–107, 2007.

[51] 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)*. ACM, 2007, pp. 16–26.

[52] E. Wong, T. Wei, Y. Qi, and L. Zhao, "A crosstab-based statistical method for effective fault localization," in *International Conference on Software Testing, Verification, and Validation((ICST 2008)*, 2008, pp. 42–51.

[53] N. Japkowicz and S. Stephen, "The class imbalance problem: A systematic study," *Intelligent Data Analysis*, vol. 6, no. 5, pp. 429–449, 2002.

[54] Y. Yu, "An empirical study of the effects of test-suite reduction on fault localization," in *ACM/IEEE International Conference on Software Engineering(ICSE 2009)*, 2009, pp. 201–210.

[55] K. Liu, A. Koyuncu, D. Kim, and T. F. Bissyandé, "Tbar: Revisiting template-based automated program repair," in *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2019, pp. 31–42.