Can This Fault Be Detected by Automated Test Generation: A Preliminary Study

Hangyuan Cheng[†], Ping Ma[†], Jingxuan Zhang[‡], Jifeng Xuan[† *]
[†] School of Computer Science
Wuhan University, Wuhan 430072, China
Email: {chenghy, pingma, jxuan}@whu.edu.cn
[‡] College of Computer Science and Technology
Nanjing University of Aeronautics and Astronautics, Nanjing 211106, China
Email: jxzhang@nuaa.edu.cn

***Abstract*—Automated test generation can reduce the manual effort to improve software quality. A test generation method employs code coverage, such as the widely-used branch coverage, to guide the inference of test cases. These test cases can be used to detect hidden faults. An automatic tool takes a specific type of code coverage as a configurable parameter. Given an automated tool of test generation, a fault may be detected by one type of code coverage, but omitted by another. In frequently released software projects, the time budget of testing is limited. Configuring code coverage for a testing tool can effectively improve the quality of projects. In this paper, we conduct a preliminary study on whether a fault can be detected by specific code coverage in automated test generation. We build predictive models with 60 metrics of faulty source code to identify detectable faults under eight types of code coverage, such as branch coverage. In the experiment, an off-the-shelf tool, EvoSuite is used to generate test data. Experimental results show that different types of code coverage result in the detection of different faults. The extracted metrics of faulty source code can be used to predict whether a fault can be detected with the given code coverage; all studied code coverage can increase the number of detected faults that are missed by the widely-used branch coverage. This study can be viewed as a preliminary result to support the configuration of code coverage in the application of automated test generation.**

***Index Terms*—test generation, code metrics, code coverage, predictive models**

I. Introduction

Automated test generation aims at automatically creating test cases to cover program paths and detect faults. Existing studies have widely investigated the techniques of test generation. For instance, Randoop by Pacheco et al. [24] is designed as a random testing method driven by feedback. Java PathFinder by Anand et al. [3] employs symbolic execution to infer test cases; EvoSuite by Fraser and Arcuri [7] uses evolutionary computation to evolve test cases to explore program paths. Applying automated test generation can improve the coverage of source code and reduce the cost of manually writing tests by developers.

Code coverage, such as branch coverage, method coverage, and mutation coverage, is a type of measurements in both manual and automated test generation. We refer to a measurement of code coverage as a *coverage measurement*. The code coverage is the ratio of executed code by running tests in all code under testing. *Branch coverage*, a widely-used code coverage, is the ratio of executed branches among all branches in source code. Tests with high code coverage are considered as "adequate" tests since empirical results show code coverage correlates with the ability of fault detection [6], [10], [15]. However, in automated test generation, different coverage measurements lead to inconsistent results of fault detection. Our experiment on test generation by EvoSuite (in Section IV-B) shows that in a time budget of 10 minutes per fault, tests based on weak mutation coverage can detect 41 faults, which are not be detected by tests based on branch coverage; meanwhile generated tests based on branch coverage can detect 54 faults that cannot be detected by generated tests based on weak mutation coverage.

For an automatic tool of test generation, a tester can configure a specific coverage measurement as a parameter, such as branch coverage or method coverage, to guide or evaluate the process of test generation. For example, in the tool EvoSuite, configuring coverage measurements results in the generation of different tests and the detection of different faults. In a limited time budget, generating tests for all coverage measurements is impractical. Instead, a tester can manually configure one or more particular coverage measurements for test generation. *In automated test generation, could we identify whether potential faults can be detected for one coverage measurement*? If the answer is positive, we can guide developers to schedule programs under testing to maximize the number of detected faults.

Existing studies on debugging have identified the effectiveness of automated techniques. Le et al. [19] proposed a predictive method to predict whether automated fault localization tools can obtain accurate results on particular faults; another work proposed by the same group [20] learned to identify whether automated program repair can generate correct patches. Motivated by the existing work [19], [20], we explore whether a fault can be detected by automated test generation based on different coverage measurements. A tester can leverage our results to choose a specific coverage measurement to increase the probability of fault detection or to schedule the test execution in a limited time budget.

*Corresponding author

In this paper, we conducted a preliminary study on the analysis of detectable faults by automated test generation. We investigated 742 real-world faulty files as well as their testing results on eight coverage measurements by an off-the-shelf test generation tool, EvoSuite. We aimed to answer three Research Questions (RQs), including the prediction of detectable faults for one coverage measurement, the difference between coverage measurements, and the metric correlation. In RQ1, we found that the classifier RandomForest with SMOTE is effective in building a predictive model of detected faults based on metrics of faulty source code. In RQ2, we found that the branch coverage can detect most faults among all coverage measurements under evaluation; the weak mutation coverage and the direct branch coverage can be used as the supplement to increase the number of detected faults. In RQ3, we analyzed which metric correlates with newly detected faults: six metrics, such as the number of public methods and the number of descendants, appear in top-5 of correlation in two groups of experiments. This study can be viewed as a preliminary result to support the identification of detectable faults with automated techniques.

This paper makes the following major contributions:

- We conducted a study on the fault detection for eight coverage measurements to assist the users to use automatic tools of test generation.
- We empirically studied the ability of fault detection by automated test generation on 742 faulty files via an automatic tool, EvoSuite.
- We designed predictive models based on 60 code metrics to learn whether a fault can be detected; we showed that the classifier of Random Forest with SMOTE is effective in the prediction.

The rest of this paper is organized as follows. Section II shows the background of this study. Section III presents the study setup, including three research questions and the data preparation. Section IV shows the results of our preliminary study. Section V presents the threats to the validity. Section VI lists the related work of the study and Section VII concludes this paper.

II. Background and Motivation

Our work aims to empirically identify whether potential faults can be detected by automated test generation. In this section, we present the background and the motivation of this work.

A. Background

In white-box testing, developers write tests to detect potential faults. A test is a piece of source code, which is formed as a test method in modern testing framework, such as JUnit. To detect faults, a developer reads the requirements of the program and then writes several tests. The requirements can be identified as *test oracle*, which validates the correctness of source code under testing [26]. The written tests are expected to be consistent with the test oracle. To trigger a fault, the source code has to be executed. Therefore, code coverage, such as branch coverage, is considered as a measurement to quantify the degree of adequate testing; a coverage measurement is also called a test adequacy criterion [31]. Branch coverage is widely-used in practice [4], [22]. To avoid any potential ambiguity, we define *fault detection* as follows. We execute a test case on a program under test. If the test execution is interrupted, including a crash or an assertion violation, we call a fault in the program is detected [29].

To reduce the manual effort of writing tests, automated techniques of test generation are proposed. Similar to writing tests by human developers, automated test generation is designed to produce tests to satisfy code coverage. However, it is difficult to directly automate the process of reading requirements. Thus, automated test generation cannot rely on software requirements that human developers can directly understand [29].

In automated test generation, a coverage measurement serves as both a measurement of evaluating automatically generated tests and a fitness function to guide the process of test generation [3], [7]. The choice of coverage measurements is a parameter in a tool of automated test generation. A user of such a tool can manually decide the configuration of coverage measurements. Ideally, automated techniques may exhaustively test all source code, but it may require an unacceptable time cost. In a limited time budget, tests generated by automated test generation techniques cannot cover all source code [10].

EvoSuite, a search-based tool, is widely-studied in automated test generation [1], [7]–[10], [15]. EvoSuite encodes a test into a chromosome of statements and employs a genetic algorithm to search for the optimal chromosome; EvoSuite iteratively generates a set of chromosomes with the guide of a fitness function, i.e., a coverage measurement. The chromosome with the best value of the fitness function is finally converted back to a generated test. In generated tests, EvoSuite embeds several types of assertions to check the program states, e.g., assertions of equal values, not null objects, and unchanged states. In the implementation of EvoSuite, many coverage measurements can be deployed, including branch coverage, line coverage, and weak mutation coverage.

B. Motivation

Different coverage measurements lead to various results of fault detection [16], [25]. Given the same running time, a fault may be detected with a coverage measurement, e.g., method coverage, but may be not detected with another coverage measurement, e.g., branch coverage.

Motivated by the above fact, in this study, we present the ability of fault detection with different coverage measurements via automated test generation and we present the feasibility of identifying whether automatically generated tests can detect a hidden fault with one coverage measurement. A user of an automatic tool of test generation can follow our study to choose a specific coverage measurement to enlarge the probability of fault detection. Meanwhile, if the time cost of testing is limited, a developer can choose to run automated

TABLE I
LIST OF METRICS OF SOURCE CODE

Category	Metric
Clone	Clone coverage, clone classes, clone complexity, clone instances, clone line coverage, clone logical line coverage, lines of duplicated code, logical lines of duplicated code
Cohesion	Lack of cohesion in methods
Complexity	Nesting level, nesting level else-if, weighted methods per class
Coupling	Coupling between object classes, coupling between object classes inverse, number of incoming invocations, number of outgoing invocations, response set for class
Documentation	API documentation, comment density, comment lines of code, documentation lines of code, public documented API, public undocumented API, total comment density, total comment lines of code
Inheritance	Depth of inheritance tree; number of ancestors, children, descendants, and parents
Size	*Direct number*(†) and *total number*(†) of attributes, getters, lines of code, logical lines of code, respectively; *direct number*(†) and *total number*(†) of local attributes(‡), local getters, local methods, local public attributes, local public methods, local setters, methods, statements, public attributes, public methods, and setters, respectively

† *Direct number* and *total number* denote counting the number without and with inherited attributes from ancestors classes, respectively.
‡ *Local* denotes the number that are defined inside nested, anonymous, and local classes.

test generation on a fault with the high likelihood of being detected. Based on this study, developers can also understand the factors that relate to the detected faults or try multiple times of test generation with different coverage measurements in limited time. We expect to provide a preliminary result to support the choice of coverage measurements in automated test generation.

III. STUDY SETUP

In this section, we describe the research questions and the data preparation in the study.

A. Research Questions

We conducted a preliminary study on whether a fault can be detected by automated test generation. This study is designed to understand detectable faults based on test generation and to answer three Research Questions (RQs).

RQ1. Can we predict whether a fault is detectable with specific code coverage? We investigate the potential of predicting detectable faults. Since multiple coverage measurements may lead to the detection of different faults. In RQ1, we focus on the prediction based on each specific coverage measurement. We are to build a classifier to identify detectable faults by automated test generation.

RQ2. How many faults can be newly detected by other coverage rather than branch coverage? Branch coverage is considered as a widely-used coverage in practice [8], [10]. If a fault cannot be detected by branch coverage but can be detected by another coverage, we refer to this fault as a *newly detected fault*. In RQ2, we study the difference of detected faults between branch coverage and other coverage. We explore whether other coverage measurements can reveal faults that cannot be detected with branch coverage.

RQ3. Which metric correlates with newly detected faults that are detected by other coverage rather than branch coverage? We investigate the factors related to newly detected faults that are not detected with branch coverage. In RQ3, we check the code metrics that correlate with the newly detected faults.

Relationship among three RQs. Our study focuses on the possibility of identifying whether a fault can be detected by automated test generation. In RQ1, we directly answer the effectiveness of identifying detectable faults via building a predictive model. RQ1 can be viewed as a simple result and empirical evidence for this study. In RQ2, we distinguish the impact of different coverage measurements on the fault detectability. For example, the branch coverage is widely-used for fault detection. We observe the newly detected faults by the other coverage rather than the branch coverage. In RQ3, we check the correlation between the newly detected faults and the metric of source code. RQ3 further answers the impact on the newly detected faults.

B. Data Preparation

To understand the fault detection by test generation, our study needs labeled data of test execution. In our study, we chose EvoSuite as the tool of test generation since it is easy to be used and deployed [7].

We implemented the experiment via Java JDK 1.7 on the top of a machine-learning framework Weka[1].

742 faulty classes under evaluation. Salahirad et al. [25] have executed EvoSuite on 742 real-world Java faulty files and collected the detailed testing results. These faults are extracted from 15 open-source projects, including Apache Commons Codec, CLI, CSV, JXPath, Lang, Math, JFreeChart, Guava, JacksonCore, JacksonDatabind, JacksonXML, Jsoup, Google Closure, Joda-Time, and Mockito. In our study, we followed their work to use all faulty files. One fault may contain faulty code in two or more Java classes. Thus, we treated each faulty class as one instance in our study since EvoSuite can specify

[1]Weka, http://www.cs.waikato.ac.nz/ml/weka/.

TABLE II
PRECISION (**P**), RECALL (**R**), F-MEASURE (**F**), AND AUC OF PREDICTING DETECTED FAULTS FOR **2-MINUTE** EXPERIMENTS

Coverage	# Faults		BayesNet + SMOTE				SVM + SMOTE				RandomForest + SMOTE				RandomForest			
	Detected	Undetected	P	R	F	AUC	P	R	F	AUC	P	R	F	AUC	P	R	F	AUC
Branch	289	453	0.724	0.711	0.717	0.761	0.578	**0.969**	0.724	0.533	**0.769**	0.843	**0.804**	**0.837**	0.582	0.526	0.553	0.717
Direct branch	264	478	0.667	0.754	0.708	0.741	0.545	**0.960**	0.695	0.537	**0.763**	0.822	**0.791**	**0.831**	0.580	0.496	0.535	0.735
Line	115	627	0.457	**0.487**	0.472	0.707	0.667	0.035	0.066	0.514	**0.682**	0.448	**0.541**	**0.820**	0.318	0.122	0.176	0.650
Exception	251	491	0.664	0.709	0.686	0.764	0.536	**0.960**	0.688	0.554	**0.752**	0.805	**0.778**	**0.844**	0.571	0.482	0.523	0.731
Method	132	610	0.447	**0.655**	0.531	0.727	0.385	0.019	0.036	0.503	**0.683**	0.572	**0.623**	**0.840**	0.390	0.242	0.299	0.705
Method w/o exception	140	602	0.490	**0.625**	0.549	0.751	0.250	0.007	0.014	0.499	**0.689**	0.554	**0.614**	**0.835**	0.400	0.200	0.267	0.690
Output	159	583	0.543	**0.752**	0.631	0.779	0.556	0.031	0.060	0.509	**0.740**	0.616	**0.672**	**0.854**	0.424	0.226	0.295	0.711
Weak mutation	259	483	0.683	0.687	0.685	0.747	0.536	**0.952**	0.686	0.535	**0.756**	0.797	**0.776**	**0.820**	0.560	0.413	0.476	0.702

TABLE III
PRECISION (**P**), RECALL (**R**), F-MEASURE (**F**), AND AUC OF PREDICTING DETECTED FAULTS FOR **10-MINUTE** EXPERIMENTS

Coverage	# Faults		BayesNet + SMOTE				SVM + SMOTE				RandomForest + SMOTE				RandomForest			
	Detected	Undetected	P	R	F	AUC	P	R	F	AUC	P	R	F	AUC	P	R	F	AUC
Branch	294	448	0.715	0.733	0.724	0.761	0.586	**0.978**	0.733	0.536	**0.746**	0.847	**0.793**	**0.814**	0.588	0.568	0.578	0.706
Direct branch	290	452	0.728	0.719	0.723	0.764	0.578	**0.978**	0.726	0.531	**0.748**	0.828	**0.786**	**0.816**	0.563	0.510	0.535	0.693
Line	99	643	0.544	**0.409**	0.467	0.731	0.000	0.000	0.000	0.498	**0.603**	0.399	**0.480**	**0.803**	0.342	0.131	0.190	0.603
Exception	272	470	0.682	0.699	0.690	0.743	0.555	**0.965**	0.705	0.535	**0.752**	0.825	**0.787**	**0.819**	0.549	0.478	0.511	0.697
Method	136	606	0.493	**0.665**	0.567	0.751	0.500	0.026	0.049	0.507	**0.711**	0.596	**0.648**	**0.847**	0.493	0.250	0.332	0.732
Method w/o exception	137	605	0.477	**0.609**	0.535	0.734	0.500	0.022	0.042	0.506	**0.720**	0.591	**0.649**	**0.850**	0.463	0.226	0.304	0.718
Output	166	576	0.548	**0.702**	0.616	0.771	0.353	0.018	0.034	0.499	**0.711**	0.593	**0.647**	**0.829**	0.457	0.259	0.331	0.709
Weak mutation	281	461	0.689	0.694	0.691	0.745	0.562	**0.956**	0.708	0.524	**0.753**	0.813	**0.782**	**0.814**	0.554	0.495	0.523	0.707

one class under testing as input. Thus, we obtained 742 faulty files (also faulty classes in Java), each of which is referred to as a *fault* for short.

2 groups of experiments. Salahirad et al. [25] have used two groups of setup: the time budget two minutes and ten minutes per class in test generation. We followed their setup and did not adding other time budgets since the cost of data collection is huge. And the time budget between two and ten minutes can increase the confidence of the data analysis in our study.

60 code metrics of class under test. In our study, each faulty class is converted into a vector of 60 code metrics. We used the collected data by Salahirad et al. [25] and followed their definition of code metrics. Table I briefly lists the code metrics of each fault class in our study.

8 coverage measurements. We listed eight coverage measurements in the study as follows: *Branch coverage* is the ratio of executed control-flow branches by tests. *Direct branch coverage* also considers branches, but only focuses on branches inside the method under testing. *Line coverage*, *exception coverage*, and *method coverage* are the ratio of executed lines of source code, trigged exceptions, and executed methods by tests, respectively. *Method without exception coverage* (*method w/o exception* for short) is similar to the method coverage, but does not consider any test that throws an exception. *Output coverage* is the ratio of mapped returned values to abstract values [2]. *Weak mutation coverage* is the ratio of killed mutants by tests, where a mutant is a slightly changed version of current source code [23].

Data collection. The test data is collected with EvoSuite. For each fault, EvoSuite is run to generate tests with one coverage measurement in a time budget, i.e., two minutes or ten minutes. In each run of EvoSuite, tests and code coverage are collected as the dataset.

IV. EXPLORATORY STUDY

We conducted a preliminary study on understanding detectable faults by automated test generation with eight coverage measurements. The three RQs are investigated as follows.

A. RQ1. Can we predict whether a fault is detectable with specific code coverage?

Given specific code coverage such as branch coverage, can we predict whether a fault can be detected by automated test generation? We viewed each fault as a vector of 60 code metrics and labeled the faults as *detected* or *undetected* according to the data of actual test execution (in Section III-B). Therefore, for one coverage measurement, we build a classifier to predict whether a fault can be detected or not. In this research question, we characterized each fault as a vector of code metrics. We note that 60 code metrics may not fully show the characteristics of a faulty program. However, extracting these metrics can partially show differences among faults.

In the study, we used three typical classifiers in the evaluation: BayesNet (a network classifier of multiple Bayesian nodes), SVM (an algorithm of Support Vector Machine with the kernel of radial basis functions), and RandomForest (an ensemble classifier of multiple decision trees). These three classifiers are combined with SMOTE. *SMOTE* is a method of

imbalanced data processing since the experimental data show the risk of data imbalance [5]. We evaluated the prediction results with 5-fold cross validation and showed the average. We measured the evaluation of predicting detected faults with four typical ways: precision, recall, F-measure, and AUC. A higher value shows the better effectiveness of the prediction.

Table II and Table III show the evaluation with eight coverage measurements in two groups of experiments: the cutoff time of test generation for each faulty file is set to two minutes and ten minutes, respectively. As shown in Table II, in the group of 2-minute experiments, the number of detected faults by test generation is fewer than the number of undetected faults; Specifically, the ratio of detected faults within 2-minute test generation ranges 15.5% to 38.9%. This provides an application scenario of imbalanced data processing, which indicates that combining SMOTE can improve the prediction result. Among the algorithms under evaluation, RandomForest with SMOTE achieved the best result than other algorithms: all coverage measurements (i.e., rows in Table II) reach the F-measure over 0.50 and four out of eight values of F-measure are over 0.77; all the AUC values are over 0.82. This result shows that the RandomForest with SMOTE can be used to predict whether a fault can be detected by test generation although the ratio of detected faults is from 15% to 40%. Meanwhile, besides RandomForest with SMOTE, BayesNet with SMOTE also performs well. Results of RandomForest with or without SMOTE show that the SMOTE method is effective in improving the imbalanced data.

The 10-minute experiments in Table III show similar results to the 2-minute experiments. RandomForest with SMOTE is the best method in predicting whether a fault can be detected. Seven out of eight values of F-measure are over 0.64; an exceptional value is 0.48 for the line coverage. The reason for this F-measure less than 0.50 is that the number of detected faults with the line coverage is 99, which leads to the ratio of detected faults to 13%. Note that for the line coverage, the number of detected faults in 10-minute experiments is lower than in 2-minute experiments. The major reason is that EvoSuite has an optimization mechanism that can reduce the number of assertions based on coverage to save the cost of running tests. Although 10-minute experiments can generate more assertions than 2-minute experiments, particular assertions that can detect faults may be reduced since the optimization in EvoSuite cannot identify which assertion can actually detect faults. From results in Table II and Table III, RandomForest with SMOTE can obtain the best prediction results among algorithms under evaluation. Most F-measure values are over 0.60; the AUC values are stable and over 0.80.

Discussion. In RQ1, the evaluation is conducted on two groups of time budgets of test generation, including two minutes or ten minutes per coverage measurement. Comparing to existing work in the prediction in software engineering, such as defect prediction [14], [30], [32], the size of datasets is not large. In our study, collecting the data of test execution is time-consuming. The collection relies on the local deployment of each faulty program [12]. We plan to conduct a study on a larger dataset or more groups of time budgets in the future.

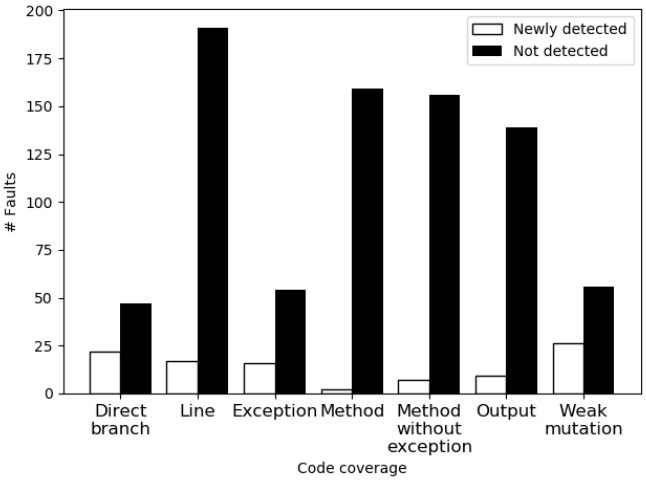

Fig. 1. Number of newly detected and not detected faults by seven other coverage measurements for 2-minute experiments, compared to branch coverage.

Finding 1. Given a coverage measurement, we can build classifiers to predict whether a fault can be detected by automated test generation. The F-measure and the AUC values show that RandomForest with SMOTE is effective in the prediction.

B. RQ2. How many faults can be newly detected by other coverage rather than branch coverage?

Branch coverage is considered as an effective coverage measurement, which is widely-used to measure both manual and automated test generation [1], [8], [15]. In Section IV-A, results in Table II and Table III show that the branch coverage can detect the most number of faults among eight coverage measurements in the study. This motivates us to investigate whether other coverage can detect a fault that is undetected by the branch coverage.

We focus on two differences between other coverage and branch coverage: for faults that cannot be detected by branch coverage, how many faults can be newly detected by other coverage? for faults that can be detected by branch coverage, how many faults are not detected by other coverage? We leverage this RQ to understand detectable faults with different coverage measurements.

Fig. 1 shows the number of newly detected and not detected faults by seven other coverage measurements for 2-minute experiments, compared with the branch coverage. For newly detected faults that the branch coverage cannot detect, the weak mutation coverage can newly detect 26 faults and the direct branch coverage can newly detect 22 faults; the method coverage, the method w/o exception coverage, and the output coverage can newly detect 2, 7, and 9 faults, respectively. This result shows that if the time budget of test generation allows using two or more coverage measurements, the weak mutation

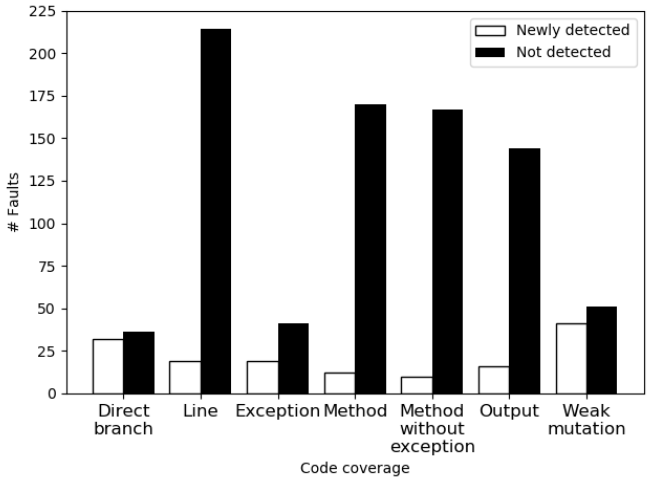

Fig. 2. Number of newly detected and not detected faults by seven other coverage measurements for 10-minute experiments, compared to branch coverage.

coverage and the direct branch coverage should be the choices while using method coverage as supplement may be not useful.

Among all detected faults by the branch coverage, the line coverage fails in detecting 191 faults; the method coverage and the method w/o exception coverage fail in detecting 159 and 156 faults, respectively. The direct branch coverage, the exception coverage, and the weak mutation coverage cannot detect 47, 54, and 56 faults, respectively, compared with the branch coverage. The result of not detected faults shows that if the line coverage is used, other coverage like branch coverage should be used as the supplement.

Fig. 2 shows the number of newly detected and not detected faults in 10-minute experiments. For newly detected faults that the branch coverage cannot detect, the weak mutation coverage can newly detect 41 faults and the direct branch coverage can newly detect 32 faults. The number of newly detected faults in 10-minute experiments are higher than that in 2-minute experiments. Among all detected faults by the branch coverage, the line coverage fails in detecting 214 faults. The number of undetected faults is also higher than that in 2-minute experiments.

Discussion. In RQ2, we show that a fault may be not detected by the widely-used branch coverage, but can be detected by another coverage, such as exception coverage. In practice of using automated test generation, a user can add several code coverage to the automatic tool to enhance the ability of fault detection via branch coverage. In this study, we did not show the correlation between the code coverage and a defect class, i.e., the type of a fault. A study on defect classes may help understand the role of coverage measurements in automated test generation.

Finding 2. Among all code coverage under evaluation, the branch coverage can detect the most faults. The weak mutation coverage and the direct branch coverage can be used as the supplement of the branch coverage and can increase newly detected faults.

C. RQ3. Which metric correlates with newly detected faults that are detected by other coverage rather than branch coverage?

In RQ2 (in Section IV-B), we showed that seven coverage measurements can be used to detect faults that are missed by the branch coverage; in RQ1 (Section IV-A), we showed that each faulty file is viewed as a vector of 60 metrics. Therefore, we plan to explore which metric correlates with the newly detected faults.

To understand the correlation, we labeled each fault with a Boolean flag, which denotes whether the fault can be newly detected by another code coverage rather than the branch coverage. We used Point-biserial correlation coefficient to measure the correlation between each metric and the flag of new fault detection. We chose Point-biserial correlation coefficient rather than other coefficients like Pearson correlation coefficient since the flag of new fault detection is a Boolean value. The Point-biserial correlation coefficient is between -1 to 1; a high absolute value indicates that the metric and the flag are highly correlative.

Given a dataset D that contains n elements, let X and Y be a continuous variable and a Boolean variable of each element. The set D is divided into two subsets D_t and D_f, where Y is true for all elements in D_t and Y is false for all elements in D_f. The Point-biserial correlation coefficient of the set D is defined as follows,

$$r_{pb} = \frac{m_t - m_f}{s} \sqrt{\frac{n_t - n_f}{n_t + n_f}}$$

where s is the standard deviation of X values in D, m_t and m_f are the mean values of X in D_t and D_f and n_t and n_f are the numbers of elements in D_t and D_f, respectively.

For each code coverage (except the branch coverage), we calculated Point-biserial correlation coefficient between each of 60 metrics and the flag of new fault detection. We sorted all correlation coefficients in a descending order of the absolute values and prioritized the correlation coefficients with statistical significance. Table IV presents the names of top-5 metrics for each code coverage. For instance, the first row in the table shows that the direct branch coverage can newly detect faults compared with the branch coverage and the top-5 metrics with the highest correlation are clone line coverage, clone coverage, clone logical line coverage, lines of duplicated code, and logical lines of duplicated code.

Table IV contains two groups of experiments: 2 minutes per fault and 10 minutes per fault of test generation. In each group, we marked metrics that appear for two and more times with the light-gray. We observed that, in the 2-minute experiments, 7 metrics appear two or more times, including clone line

TABLE IV
TOP-5 METRICS WITH THE STRONGEST CORRELATION WITH NEWLY DETECTED FAULTS BASED ON POINT-BISERIAL CORRELATION COEFFICIENT. A LIGHT-GRAY CELL SHOWS THAT THE METRIC APPEARS TWICE OR MORE AMONG SEVEN MEASUREMENTS OF CODE COVERAGE.

Coverage	Top-5 correlated metrics				
	The 1st metric	The 2nd metric	The 3rd metric	The 4th metric	The 5th metric
Group of **2-minute** experiments					
Direct branch	Clone line coverage*	Clone coverage*	Clone logical line coverage*	Lines of duplicated code	Logical lines of duplicated code
Line	Clone line coverage	Clone coverage	Clone logical line coverage	Number of public methods	Number of methods
Exception	Lack of cohesion in methods*	Number of descendants	Number of local getters	Number of outgoing invocations	Number of incoming invocations
Method	Total number of setters*	Total number of methods*	Number of local attributes*	Total number of local setters*	Number of local setters*
Method w/o exception	Number of descendants*	Total number of setters*	Total number of methods*	Number of local public attributes	Number of local setters
Output	Number of descendants	Clone coverage	Clone logical line coverage	Clone line coverage	Coupling between object classes
Weak mutation	Number of descendants*	Clone coverage*	Clone line coverage*	Clone logical line coverage*	Number of children*
Group of **10-minute** experiments					
Direct branch	Documentation lines of code*	Public documented API*	Number of local public methods*	Comment lines of code	Total comment lines of code
Line	Number of attributes	Number of descendants	Number of public methods	Total comment lines of code	Number of local public methods
Exception	Number of public attributes*	Total number of public attributes	Nesting level else-if	Number of attributes	Total number of local attributes
Method	Number of getters*	Total number of getters*	Number of local getters*	Total number of setters*	Total number of methods*
Method w/o exception	Number of getters*	Number of local getters*	Total number of getters*	Total number of methods*	Number of methods*
Output	Number of getters*	Number of public methods*	Number of local getters*	Total comment lines of code*	Documentation lines of code*
Weak mutation	Total number of public attributes*	Number of public attributes*	Nesting level else-if*	Number of getters	Number of attributes

* denotes the correlation coefficient is statistically significant (p-value< 0.05).

coverage, clone coverage, etc. In the 10-minute experiments, 13 metrics appear two or more times, including documentation lines of code, nesting level else-if, etc. The intersection of metrics in two groups consists of six metrics: the number of public methods, the number of methods, the number of descendants, the number of local getters, the total number of setters, and the total number of methods. This observation shows that these metrics play an important role in identifying newly detected faults by other coverage measurements rather than the branch coverage.

Discussion. The correlation coefficient in this study is calculated with Point-biserial correlation coefficient. Other analysis that involves the correlation between metrics, such as ANOVA [14], can lead to accurate analytical results in the correlation coefficient.

Finding 3. We analyzed which metric correlates with newly detected faults with seven other coverage measurements, compared with the branch coverage. Six metrics, such as the number of public methods and the number of descendants, appear in the top-5 correlation in both two groups of experiments.

V. THREATS TO VALIDITY

We present threats to the validity of our study in three categories.

Threats to construct validity. Our study used the dataset by Salahirad et al. [25] to study the ability of fault detection by automated test generation with different coverage measurements. Collecting data of test execution is time-consuming. This dataset contains the test results with eight coverage measurements in two groups. One group is set to two minutes per fault and the other is set to ten minutes per fault. There exists a threat that data of diverse time costs may lead to different experimental results. In Section IV-B, we answered RQ2 via showing newly detected faults and missed faults between different coverage measurements. The reason for this difference has not been explored. A study with manual analysis can provide further details. In the study, we only use one tool of test generation, i.e., EvoSuite. This adds a threat to the generality of the result. A study on other tools of test generation can be conducted to check the generality.

Threats to internal validity. In Section IV-A, we conducted a predictive model for the detection of each fault. The number of instances in the dataset may be not enough for building a stable predictive model. In the evaluation, we used available data of our dataset of eight coverage measurements. A large scale evaluation could be explain the benefit of predicting the fault detection.

Threats to external validity. The generality is a threat to applying the empirical results and findings of our study. Our dataset contains 742 faulty files of Java programs. We do not claim that this preliminary study on detectable faults can be generalized to other projects, other test scenarios, or other automated tools of test generation. Considering the number of test generation in daily development, a study on a diversified dataset can help to understand the detected faults by various coverage measurements.

VI. RELATED WORK

Automated test generation has been widely-studied and many automated tools are proposed, including Randoop by Pacheco et al. [24], Java PathFinder by Anand et al. [3], and EvoSuite by Fraser and Arcuri [7]. Among existing tools of test generation, EvoSuite is used to detect faults or generate specific tests due to its high usability. Fraser and Arcuri [10] designed a large scale experiment of 1600 faults from 100 projects. Their experiment shows that EvoSuite can achieve high coverage and effectively detect faults. Kochhar et al. [16] conducted an empirical study on the test adequacy in 300 open-source projects. Just et al. [15] studied the usefulness of mutation coverage in detection real faults.

Automated tools of testing as well as debugging can reduce the manual effort by developers. However, the effectiveness of automated tools is highly concerned. Existing studies have explored the reliability of automated tools and tended to identify the feasibility of applying these tools in practice. Le et al. [17], [18] designed an predictive method on the effectiveness of fault localization, which ranks suspicious source code to assist debugging. Le et al. [20] proposed to identify the feasibility of applying automated program repair techniques to generate patches. Jiang et al. [13] proposed to identify the cause of test alarms in the integration testing of real systems. Grano et al. [11] developed machine-learning methods to predict the coverage by test cases. Xu et al. [28] characterized higher-order functions in Scala language to prioritize functions that should be tested. Le et al. [19] designed a method to identify the reliability of bug localization, which recommends related source files to a given bug report. Li et al. [21] proposed a recommendation method to rank exception handling strategies for potential exceptions in source code. Gu et al. [12] designed a predictive method for crashing fault residence to identify whether the root cause of a crash resides in the stack trace.

Tantithamthavorn and Hassan [27] have conducted an experience report on the pitfalls and the opportunities for practical defect prediction. Jiarpakdee et al. [14] presented the impact of correlation between metrics in the predictive models on defect prediction.

In the study of test generation, Salahirad et al. [25] proposed a controlled experiment to study the impacts among source metrics, test metrics, and code coverage. Their study provided detailed analysis for whether achieving high coverage can effectively detect real faults. In this paper, we followed the collected dataset by Salahirad et al. [25]. Different from the existing work, our study shows that it is feasible to identify detectable faults with given coverage measurements; we further investigate the correlation between metrics of faulty code and newly detected faults; our study also illustrated that the choice of code coverage to minimize the number of undetected faults within a limited time budget.

VII. Conclusions

We conducted a preliminary study on whether a fault can be detected by automated test generation. We investigated detected faults on a dataset of 742 faulty files with test generation in two minutes and ten minutes per faults. Experimental results show that RandomForest with SMOTE can achieve effective prediction on detected faults for all coverage measurements; the results also show that in a limited time budget of test generation, trying multiple coverage measurements leads to more detected faults than using one single type of code coverage.

In future work, we plan to explore the reason of different detectable faults with a large dataset of test execution. Such exploration can provide a deep explanation for the factors of fault detectability. We also plan to analyze the metrics of faulty code to further understand potential relationship among metrics, faults, and tests.

Acknowledgment

The authors sincerely thank Alireza Salahirad, Hussein Almulla, and Gregory Gay for sharing their testing data. This work is supported by the National Key R&D Program of China under Grant No. 2018YFB1003901, the National Natural Science Foundation of China under Grant No. 61872273, the Open Research Fund Program of CETC Key Laboratory of Aerospace Information Applications under Grant No. SXX18629T022, and the Advance Research Projects of Civil Aerospace Technology, Intelligent Distribution Technology of Domestic Satellite Information under Grant No. B0301.

References

[1] M. M. Almasi, H. Hemmati, G. Fraser, and A. Arcuri. An industrial evaluation of unit test generation: Finding real faults in a financial application. In *39th International Conference on Software Engineering, ICSE 2017, Software Engineering in Practice Track*, pages 263–272. IEEE, 2017.

[2] N. Alshahwan and M. Harman. Coverage and fault detection of the output-uniqueness test selection criteria. In *International Symposium on Software Testing and Analysis, ISSTA '14, San Jose, CA, USA - July 21 - 26, 2014*, pages 181–192, 2014.

[3] S. Anand, C. S. Pasareanu, and W. Visser. JPF-SE: A symbolic execution extension to java pathfinder. In *Proceedings of 13th International Conference on Tools and Algorithms for the Construction and Analysis of Systems, , TACAS 2007*, pages 134–138, 2007.

[4] V. Blondeau, A. Etien, N. Anquetil, S. Cresson, P. Croisy, and S. Ducasse. What are the testing habits of developers? A case study in a large IT company. In *2017 IEEE International Conference on Software Maintenance and Evolution, ICSME 2017, Shanghai, China, September 17-22, 2017*, pages 58–68, 2017.

[5] N. V. Chawla, K. W. Bowyer, L. O. Hall, and W. P. Kegelmeyer. SMOTE: synthetic minority over-sampling technique. *J. Artif. Intell. Res.*, 16:321–357, 2002.

[6] T. T. Chekam, M. Papadakis, Y. L. Traon, and M. Harman. An empirical study on mutation, statement and branch coverage fault revelation that avoids the unreliable clean program assumption. In *Proceedings of the 39th International Conference on Software Engineering, ICSE 2017, Buenos Aires, Argentina, May 20-28, 2017*, pages 597–608, 2017.

[7] G. Fraser and A. Arcuri. Evosuite: automatic test suite generation for object-oriented software. In *SIGSOFT/FSE'11 19th ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE-19) and ESEC'11: 13th European Software Engineering Conference (ESEC-13), Szeged, Hungary, September 5-9, 2011*, pages 416–419, 2011.

[8] G. Fraser and A. Arcuri. Sound empirical evidence in software testing. In *34th International Conference on Software Engineering, ICSE 2012, June 2-9, 2012, Zurich, Switzerland*, pages 178–188. IEEE, 2012.

[9] G. Fraser and A. Arcuri. A large scale evaluation of automated unit test generation using evosuite. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 24(2):8, 2014.

[10] G. Fraser and A. Arcuri. 1600 faults in 100 projects: automatically finding faults while achieving high coverage with evosuite. *Empirical Software Engineering*, 20(3):611–639, 2015.

[11] G. Grano, T. V. Titov, S. Panichella, and H. C. Gall. Branch coverage prediction in automated testing. *Journal of Software: Evolution and Process*, 2019.

[12] Y. Gu, J. Xuan, H. Zhang, L. Zhang, Q. Fan, X. Xie, and T. Qian. Does the fault reside in a stack trace? assisting crash localization by predicting crashing fault residence. *Journal of Systems and Software*, 148:88–104, 2019.

[13] H. Jiang, X. Li, Z. Yang, and J. Xuan. What causes my test alarm?: automatic cause analysis for test alarms in system and integration testing. In *Proceedings of the 39th International Conference on Software Engineering, ICSE 2017, May 20-28, 2017*, pages 712–723, 2017.

[14] J. Jiarpakdee, C. Tantithamthavorn, and A. E. Hassan. The impact of correlated metrics on the interpretation of defect models. *IEEE Transactions on Software Engineering*, to appear.

[15] R. Just, D. Jalali, L. Inozemtseva, M. D. Ernst, R. Holmes, and G. Fraser. Are mutants a valid substitute for real faults in software testing? In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering, (FSE-22), Hong Kong, China, November 16 - 22, 2014*, pages 654–665, 2014.

[16] P. S. Kochhar, F. Thung, D. Lo, and J. L. Lawall. An empirical study on the adequacy of testing in open source projects. In *21st Asia-Pacific Software Engineering Conference, APSEC 2014, Jeju, South Korea, December 1-4, 2014. Volume 1: Research Papers*, pages 215–222, 2014.

[17] T. B. Le and D. Lo. Will fault localization work for these failures? an automated approach to predict effectiveness of fault localization tools. In *2013 IEEE International Conference on Software Maintenance, September 22-28, 2013*, pages 310–319, 2013.

[18] T. B. Le, D. Lo, and F. Thung. Should I follow this fault localization tool's output? - automated prediction of fault localization effectiveness. *Empirical Software Engineering*, 20(5):1237–1274, 2015.

[19] T. B. Le, F. Thung, and D. Lo. Will this localization tool be effective for this bug? mitigating the impact of unreliability of information retrieval based bug localization tools. *Empirical Software Engineering*, 22(4):2237–2279, 2017.

[20] X. D. Le, T. B. Le, and D. Lo. Should fixing these failures be delegated to automated program repair? In *26th IEEE International Symposium on Software Reliability Engineering, ISSRE 2015, Gaithersbury, MD, USA, November 2-5, 2015*, pages 427–437, 2015.

[21] Y. Li, S. Ying, X. Jia, Y. Xu, L. Zhao, G. Cheng, B. Wang, and J. Xuan. Eh-recommender: Recommending exception handling strategies based on program context. In *23rd International Conference on Engineering of Complex Computer Systems, ICECCS 2018, Melbourne, Australia, December 12-14, 2018*, pages 104–114. IEEE Computer Society, 2018.

[22] D. B. Martin, J. Rooksby, M. Rouncefield, and I. Sommerville. 'Good' organisational reasons for 'bad' software testing: An ethnographic study of testing in a small software company. In *29th International Conference on Software Engineering (ICSE 2007), Minneapolis, MN, USA, May 20-26, 2007*, pages 602–611, 2007.

[23] A. S. Namin and S. Kakarla. The use of mutation in testing experiments and its sensitivity to external threats. In *Proceedings of the 20th International Symposium on Software Testing and Analysis, ISSTA 2011, Toronto, ON, Canada, July 17-21*, pages 342–352, 2011.

[24] C. Pacheco, S. K. Lahiri, M. D. Ernst, and T. Ball. Feedback-directed random test generation. In *29th International Conference on Software Engineering (ICSE 2007), May 20-26, 2007*, pages 75–84, 2007.

[25] A. Salahirad, H. Almulla, and G. Gay. Choosing the fitness function for the job: Automated generation of test suites that detect real faults. *Software Testing, Verification and Reliability*, 29(4-5):to appear, 2019.

[26] M. Staats, G. Gay, and M. P. E. Heimdahl. Automated oracle creation support, or: How I learned to stop worrying about fault propagation and love mutation testing. In *34th International Conference on Software Engineering, ICSE 2012, June 2-9, 2012*, pages 870–880, 2012.

[27] C. Tantithamthavorn and A. E. Hassan. An experience report on defect modelling in practice: pitfalls and challenges. In F. Paulisch and J. Bosch, editors, *Proceedings of the 40th International Conference on Software Engineering: Software Engineering in Practice, ICSE (SEIP) 2018, Gothenburg, Sweden, May 27 - June 03, 2018*, pages 286–295. ACM, 2018.

[28] Y. Xu, X. Jia, and J. Xuan. Writing tests for this higher-order function first: Automatically identifying future callings to assist testers. In *Proceedings of the 11th Asia-Pacific Symposium on Internetware*, Internetware, 2019.

[29] J. Xuan and M. Monperrus. Test case purification for improving fault localization. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering, (FSE-22), Hong Kong, China, November 16 - 22*, pages 52–63, 2014.

[30] F. Zhang, Q. Zheng, Y. Zou, and A. E. Hassan. Cross-project defect prediction using a connectivity-based unsupervised classifier. In *Proceedings of the 38th International Conference on Software Engineering, ICSE 2016, Austin, TX, USA, May 14-22, 2016*, pages 309–320, 2016.

[31] H. Zhu, P. A. V. Hall, and J. H. R. May. Software unit test coverage and adequacy. *ACM Comput. Surv.*, 29(4):366–427, 1997.

[32] T. Zimmermann, N. Nagappan, H. C. Gall, E. Giger, and B. Murphy. Cross-project defect prediction: a large scale experiment on data vs. domain vs. process. In *Proceedings of the 7th joint meeting of the European Software Engineering Conference and the ACM SIGSOFT International Symposium on Foundations of Software Engineering, 2009, Amsterdam, The Netherlands, August 24-28, 2009*, pages 91–100, 2009.