The Journal of Systems & Software 170 (2020) 110769

Contents lists available at ScienceDirect

The Journal of Systems & Software

journal homepage: www.elsevier.com/locate/jss

Can this fault be detected: A study on fault detection via automated test generation

Ping Ma [a], Hangyuan Cheng [a], Jingxuan Zhang [b], Jifeng Xuan [a,*]

[a] School of Computer Science, Wuhan University, Wuhan 430072, China
[b] College of Computer Science and Technology, Nanjing University of Aeronautics and Astronautics, Nanjing 211106, China

ARTICLE INFO

Article history:
Received 16 April 2020
Received in revised form 21 July 2020
Accepted 30 July 2020
Available online 20 August 2020

Keywords:
Test generation
Code metrics
Code coverage
Predictive models

ABSTRACT

Automated test generation can reduce the manual effort in improving software quality. A test generation method employs code coverage, such as the widely-used branch coverage, to guide the inference of tests. These tests 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 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. In the experiment, an off-the-shelf tool, EvoSuite is used to generate test data. Experimental results based on four research questions show that different types of code coverage result in the detection of different faults; a code coverage can be used as a supplement to increase the number of detected faults if another coverage is applied first; for each coverage, the number of detected faults increases with its cutoff time in test generation. Our result shows that the choice of code coverage can be learned via multi-objective optimization from sampled faults and directly applied to new faults. This study can be viewed as a preliminary result to support the configuration of code coverage in the application of automated test generation.

1. Introduction

Tests are designed to execute program paths and detect faults during program execution. Automated test generation is a technique that helps developers automatically create tests. Many researchers have investigated the techniques of automated test generation. For instance, Randoop by Pacheco et al. (2007) is developed as a random testing method driven by feedback; Java PathFinder by Anand et al. (2007) employs symbolic execution to infer tests; EvoSuite by Fraser and Arcuri (2011) uses evolutionary computation to evolve tests to explore program paths. Developers can use automated test generation to improve the coverage of source code and reduce the cost of manually writing tests.

Code coverage is used to measure the program execution of a test on the program under test. Typical code coverage, such as branch coverage, line coverage, or method coverage, is defined as the ratio of executed program elements (such as branches, lines, and methods) by tests that are manually written or automatically generated. We refer to a measurement of code coverage as a *coverage criterion*. Among these coverage criteria, *branch coverage*, a widely-used code coverage, is the ratio of executed branches among all branches in source code. We consider a test with high code coverage as an "adequate" test since empirical results show that code coverage correlates with the ability of fault detection (Fraser and Arcuri, 2015; Chekam et al., 2017; Just et al., 2014; Xuan et al., 2017).

In a tool of automated test generation, coverage criterion is a parameter that can be configured by testers. The coverage criterion can guide the generation of different tests. For example, in the tool EvoSuite, the search process of tests is conducted based on the configuration of coverage criteria. The choice of coverage criteria results in different tests and then causes various results of fault detection. For instance, our experiment on test generation by EvoSuite (in Section 4.2) shows that in a time budget of 10 min per fault, tests based on weak mutation coverage can detect 41 faults that are undetected by tests based on branch coverage. Meanwhile, tests based on branch coverage can detect 54 faults that cannot be detected by tests based on weak mutation coverage.

In a limited time budget, generating tests for all coverage criteria is impractical. A tester needs to make a decision among

* Corresponding author.
E-mail address: jxuan@whu.edu.cn (J. Xuan).

https://doi.org/10.1016/j.jss.2020.110769
0164-1212/

different coverage criteria to complete test generation in a limited time budget. In general, these tests are expected to detect the maximum number of faults. Then, the tester manually configures one or more particular coverage criteria for test generation. *In automated test generation, could we identify whether potential faults can be detected for one coverage criterion*? *Could we identify which coverage criteria should be configured*? If the answer is positive, we can guide developers to schedule test generation for programs under testing to maximize the number of detected faults.

Existing studies on debugging have identified the effectiveness of automated techniques. Le et al. (2017) 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 (Le et al., 2015a) learned to identify whether automated program repair can generate correct patches. Motivated by the existing work (Le et al., 2017, 2015a), we explore whether a fault can be detected by automated test generation based on different coverage criteria. A tester can leverage our results to choose a specific coverage criterion to increase the probability of fault detection or to schedule the test generation in a limited time budget.

In this paper, we conducted a 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 criteria by an off-the-shelf test generation tool, EvoSuite. We aimed to answer four Research Questions (RQs), including the prediction of detectable faults for one coverage criterion, the newly detected faults if a coverage criterion is applied after another, the evolution of fault detection via increasing the time cost, and the learnable configuration of coverage criteria from sampled faults. 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. We analyzed which metric correlates with the prediction of fault detection: twelve metrics like API documentation, the number of setters, and the number of parents, appear in top-5 of correlation in two groups of experiments. In RQ2, we found that the branch coverage can detect most faults among all coverage criteria 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, the number of detected faults evolves via changing the setting of time cost per fault. In RQ4, we converted the problem of coverage criterion selection into a problem of multi-objective optimization. This optimization problem is to minimize the number of undetected faults and the time cost of test generation per fault. The solution to the problem is trained from sampled faults and is applied to new faults. This study can be viewed as a result to support the identification of detectable faults with automated techniques.

This paper is an extension of our previous work (Cheng et al., 2020). The extension adds new empirical results on the analysis of applying one coverage criterion after another, the exploration of detected faults with different time costs, and the learnable configuration of coverage criteria from sampled faults in a limited time budget.

This paper makes the following major contributions:

- We conducted a study on the fault detection for eight coverage criteria to assist testers to configure 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.
- We proposed a multi-objective optimization model to choose coverage criteria in a limited time budget. This optimization model can be used to minimize the number of undetected faults via learning from sampled faults.

The rest of this paper is organized as follows. Section 2 shows the background of this study. Section 3 presents the study setup, including four research questions and data preparation. Section 4 shows the experimental results of our study. Section 5 presents the threats to the validity. Section 6 lists the related work of the study and Section 7 concludes this paper.

2. 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.

2.1. 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 a 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 a *test oracle*, which validates the correctness of source code under testing (Staats et al., 2012). 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 criterion is also called a test adequacy criterion (Zhu et al., 1997). Branch coverage is widely-used in practice (Martin et al., 2007; Blondeau et al., 2017). To avoid any potential ambiguity, we define *fault detection* as follows. We execute a test 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.

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 (Xuan and Monperrus, 2014). In automated test generation, a coverage criterion serves as both a measurement of evaluating automatically generated tests and a fitness function to guide the process of test generation (Anand et al., 2007; Fraser and Arcuri, 2011). The choice of coverage criteria is a parameter in a tool of automated test generation. A user of such a tool can manually decide the configuration of coverage criteria. 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 (Fraser and Arcuri, 2015).

EvoSuite, a search-based tool, is widely-studied in automated test generation (Fraser and Arcuri, 2015, 2011; Just et al., 2014; Fraser and Arcuri, 2014; Almasi et al., 2017; Fraser and Arcuri, 2012). The input of EvoSuite is a class under test and a coverage criterion; the output is a set of test classes. 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 criterion. 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

```
149  public void add(BoxAndWhiskerItem item, Comparable rowKey, Comparable columnKey) {
150    this.data.addObject(item, rowKey, columnKey);
151    int r = this.data.getRowIndex(rowKey);
152    int c = this.data.getColumnIndex(columnKey);
153    if ((this.maximumRangeValueRow == r && this.maximumRangeValueColumn == c)
154        || (this.minimumRangeValueRow == r && this.minimumRangeValueColumn == c)) {
155      updateBounds();
156    }
157  + else {
158      double minval = Double.NaN;
159      if (item.getMinOutlier() != null) {
160        minval = item.getMinOutlier().doubleValue();
161      }
162      [ ... ]
163  + }
164    fireDatasetChanged();
165  }
```

Fig. 1. Excerpt of `add()` in class `DefaultBoxAndWhiskerCategoryDataset` under package `org.jfree.data.statistics` in Project JFreeChart.

equal values, not null objects, and unchanged states. In the implementation of EvoSuite, many coverage criteria can be deployed, including branch coverage, line coverage, and weak mutation coverage.

2.2. Motivation

Different coverage criteria lead to various results of fault detection (Kochhar et al., 2014; Salahirad et al., 2019). Given the same running time, a fault may be detected with a coverage criterion, e.g., method coverage, but may be not detected with another coverage criterion, e.g., branch coverage.

Fig. 1 shows an excerpt of 17 lines of code in a method `add()` in class `org.jfree.data.statistics.DefaultBoxAndWhiskerCategoryDataset` in Project JFreeChart.[1] The method `add()` is designed to add a list of values relating to a box and whisker entity to a target table. The fault is caused by the lack of embedding the source code in an `else` branch. We apply EvoSuite to generate tests for this class. If we choose the branch coverage and set the time budget to two minutes, the tests generated by EvoSuite cannot detect the fault in Fig. 1. If we choose the direct branch coverage instead, then the fault can be detected by the generated tests. *Direct branch coverage* counts the coverage of branches through direct method calls while branch coverage considers both direct and indirect method calls (Gay, 2018; Rojas et al., 2015). The fault detection in Fig. 1 suggests that the direct branch coverage can add newly detected faults if it is applied after the branch coverage in test generation.

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

[1] Project JFreeChart, http://www.jfree.org/jfreechart/.

3. Study setup

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

3.1. Research questions

We aim to conduct a study on whether a fault can be detected by automated test generation. This study is designed to explore detectable faults based on test generation and to answer four Research Questions (RQs).

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

RQ2. How many faults can be newly detected if one coverage criterion is applied after another? Branch coverage is considered as a widely-used coverage in practice (Fraser and Arcuri, 2012, 2015). 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* when a coverage criterion is applied after branch coverage. In RQ2, we study the difference of detected faults between branch coverage and other coverage. We explore whether other coverage criteria can reveal faults that cannot be detected with branch coverage.

RQ3. How many faults can be newly detected by increasing the time of test generation from two minutes to ten? We investigate the evolution of detected fault via changing the time cost of test generation per fault. We examined the increased number of detected faults between a group of the time unit of two minutes and another group of ten minutes. Meanwhile, we examined the change by increasing time units.

RQ4. Can we configure coverage criteria via learning from sampled faults? In automated test generation, two different coverage criteria cannot detect the same faults. Thus, trying more

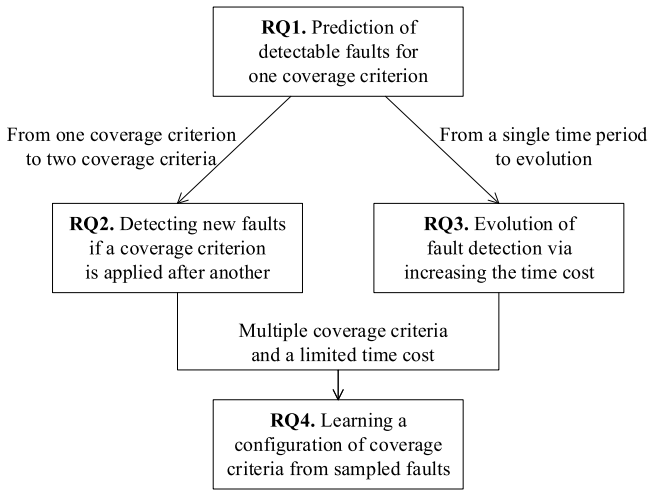


Fig. 2. Relationship among four RQs.

than one coverage criterion may increase the number of fault detection. Given a limited time budget, is it possible to select multiple coverage criteria to reduce the number of undetected faults? In RQ4, we expect that the experimental result can be used to assist the selection of code coverage in automated test generation.

Relationship among four RQs. Our study focuses on the possibility of identifying whether a fault can be detected by automated test generation. In RQ1, we directly show 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 criteria on the fault detectability. For example, the branch coverage is widely-used for fault detection. We observe the newly detected faults if another coverage is applied after the branch coverage. In RQ3, we check the evolution of detected faults via changing the time cost of test generation. The fault detection relies on the choice of coverage criteria. In RQ4, we examine the number of undetected faults and the time cost for the combination of coverage criteria. We use RQ4 to show the potential of learning the configuration of coverage criteria from sampled faults.

Fig. 2 shows the relationship among four RQs. From RQ1 to RQ2, we expanded from investigating the detection of faults with one coverage criterion to investigating the detection with two coverage criteria. From RQ1 to RQ3, we expanded from investigating the fault detection within one fixed time unit to investigating the evolution within multiple time units. RQ4 follows both RQ2 and RQ3. RQ4 is to further explore the impact of multiple coverage criteria and multiple time units on fault detection.

3.2. Data preparation

We implemented the experiment via Java JDK 1.7 on the top of a machine-learning framework Weka[2] and a multi-objective optimization framework MOEA.[3] We followed the default parameter settings of algorithms in these frameworks. Experimental results in this study are publicly available.[4] To understand the fault detection by test generation, our study needs labeled data of test execution. In our study, we chose the dataset by Salahirad et al. (2019). This dataset is collected from test execution of EvoSuite (Fraser and Arcuri, 2011) based on multiple configurations.

742 faulty classes under evaluation. Salahirad et al. (2019) 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 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. (2019) have used two groups of setup: the time budget of two minutes and ten minutes per class in test generation. We followed their setup and did not add other settings of the time budget since the cost of data collection is huge. In RQ3, we show the influence on detected faults when accumulating the time cost. The analysis on the time budget of two minutes and ten minutes can reveal the potential capability of fault detection with different settings of the time cost.

60 code metrics of classes 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. (2019) and followed their definition of code metrics. Table 1 briefly lists the code metrics of each fault class in our study. The *clone* metrics are used to denote the number of type-2 clones in a class. For example, clone coverage is the ratio between duplicate lines of code in a class and all lines of code in the class while the clone logical line coverage is the same measurement as clone coverage but includes the empty and comment lines. The *cohesion* metrics measure the degree of cohesion in a class. Specifically, the lack of cohesion in methods computes the number of coherent classes. The *complexity* metrics measure the complexity of a class. For example, the nesting level computes the depth of nesting through methods. The *coupling* metrics describe the degree of dependency between two classes. Developers expect as little dependency as possible between two classes. The *documentation* metrics represent how well a class is documented by developers. For example, the comment lines of code indicate the lines of source code that are commented. The *inheritance* metrics represent the inheritance structure of a class and the depth of the inheritance tree indicates the number of classes from the class to its furthest ancestor. The *size* metrics represent the size of a class for the structural elements, such as lines of code and local setters.

8 coverage criteria. Table 2 lists eight coverage criteria in this study. *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 (Rojas et al., 2015). *Line coverage*, *exception coverage*, and *method coverage* are the ratios of executed lines of source code, triggerd 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 (Alshahwan and Harman, 2014). *Weak mutation coverage* is the ratio of killed mutants by tests, where a mutant is a slightly changed version of current source code (Namin and Kakarla, 2011).

5 multi-objective optimization algorithms. We used five well-known multi-objective optimization algorithms, including Non-dominated Sorting Genetic Algorithm II (NSGA-II) (Deb et al.,

[2] Weka, http://www.cs.waikato.ac.nz/ml/weka/.
[3] MOEA, http://moeaframework.org/.
[4] Experimental results, http://cstar.whu.edu.cn/p/fault-detectable/.

Table 1
List of source code metrics in seven categories.

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 of items that is defined inside nested, anonymous, and local classes.

Table 2
Description of eight coverage criteria.

Coverage criterion	Description
Branch coverage	Ratio of executed control-flow branches by tests.
Direct branch coverage	Ratio of executed control-flow branches by tests without considering branches in indirect method calls.
Line coverage	Ratio of executed lines of source code by tests.
Exception coverage	Ratio of executed lines of trigged exceptions by tests.
Method coverage	Ratio of executed methods by tests.
Method without exception coverage	Ratio of executed methods by tests without considering the methods that throw exceptions.
Output coverage	Ratio of mapped returned values to abstract values.
Weak mutation coverage	Ratio of killed mutants by tests. A mutant is a slightly changed version of current source code.

2002), Decomposition-Based Evolutionary Algorithm (DBEA) (Asafuddoula et al., 2015), Dominance relation-based Multi-Objective Evolutionary Algorithm (e-MOEA) (Sun et al., 2019), Reference-Point-Based Nondominated Sorting Genetic Algorithm III (NSGA-III) (Deb and Jain, 2014), and Indicator-Based Evolutionary Algorithm (IBEA) (Yuan et al., 2016). If two objectives are to be optimized, multi-objective optimization finds out the *dominance* between solutions, which indicates the relationship that one solution is better than another. In each of five algorithms, the number of iterative generations is set to 100000 and the population is set to 100.

Data collection. The test data is collected via running EvoSuite. For each fault, EvoSuite is run to generate tests with one coverage criterion 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. The detailed setup of running EvoSuite can be found in Salahirad et al. (2019).

Evaluation for prediction. We measured the evaluation of predicting detected faults with four typical measurements: precision, recall, F-measure, and AUC. These measurements are defined based on True Positive (TP), False Positive (FP), False Negative (FN) as follows,

- TP: # of detected faults that are predicted as detected;
- FP: # of undetected faults that are predicted as detected;
- FN: # of detected faults that are predicted as undetected.

Then we defined the measurements in the evaluation of prediction as follows,

$$Precision = \frac{TP}{TP + FP}$$

$$Recall = \frac{TP}{TP + FN}$$

$$F\text{-}measure = \frac{2 \times Precision \times Recall}{Precision + Recall}$$

where the precision is the proportion of returned results that are truly correct; the recall is the proportion of the truly correct number in all retrieved results in the test set; the F-measure is the trade-off between precision and recall. The other measurement is the AUC, which is a probability value that places a positive sample prior to a negative sample (Hall et al., 2009). A higher value shows the better effectiveness of the prediction.

4. Exploratory study

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

4.1. RQ1. Can we predict whether a fault is detectable with a specific coverage criterion ?

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 3.2). Therefore, for one coverage criterion, 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 (Chawla et al., 2002). We set the number of nearest neighbors to 5 and set the random seed to default in SMOTE.

We evaluated the prediction with 5 times 5-fold cross validation and showed the average. In the 5-fold cross validation, the dataset is randomly divided into 5 equal-sized folds. Then the evaluation consists of 5 rounds; in each round, four folds are used as the training set and the other fold is used as the test set. The 5-fold cross validation is repeated for 5 times and the average is counted. We measured the evaluation of predicting detected faults with four typical ways: precision, recall, F-measure, and AUC (in Section 3.2).

Tables 3 and 4 show the evaluation with eight coverage criteria 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 3, 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 criteria (i.e., rows in Table 3) 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 4 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 the results in Tables 3 and 4, 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.

We employed the multivariate lasso regression analysis to further show the correlation between fault detection and 60 code metrics. The *multivariate lasso regression analysis* is a regression analysis for multiple variables that may contain potential dependency (Walpole et al., 2007). We did not choose Pearson correlation coefficient analysis since the independent variable (whether the fault is detectable) is binary; we did not choose multivariate logistic regression analysis since the dataset in our study (i.e., 742 faults with 60 metrics) may lead to the issue of invertible matrices. The absolute values of coefficients in the multivariate lasso regression analysis indicate the correlation between the independent variable and dependent variables.

Table 5 shows Top-5 metrics with the highest correlation of detected faults based on multivariate lasso regression analysis. For each coverage criterion, we employed the multivariate lasso regression analysis to show the correlation between each of 60 metrics and the detectable faults. We sorted all correlation coefficients in a descending order of the absolute values and presented the top-5 metrics for each coverage criterion. For instance, the first line in the table shows that the top-5 metrics for the branch coverage with the highest correlations are API documentation, the number of parents, the number of ancestors, the number of local setters, and the depth of the inheritance tree.

Table 5 consists of two groups of experiments: 2 min of test generation per fault and 10 min per fault. In each group, we marked metrics that appear for two or more times with the light-gray. We observed that, in the 2-minute experiments, 11 metrics appear two or more times, including API documentation, the number of ancestors, the number of setters. The metric of the number of ancestors appears six times in total. In the 10-minute experiments, 17 metrics appear two or more times, including API documentation, clone classes, the number of parents. The intersection of metrics in two groups consists of 12 metrics, such as API documentation, the number of ancestors, the number of parents, the number of setters. This observation shows that these metrics play an important role in identifying detected faults.

Discussion. In RQ1, the evaluation is conducted on two groups of cutoff time of test generation, including two minutes or ten minutes per coverage criterion. Comparing to existing work in the prediction in software engineering, such as defect prediction (Zimmermann et al., 2009; Zhang et al., 2016; Jiarpakdee et al., 2020), the size of the dataset 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 (Gu et al., 2019; Rojas et al., 2015). We plan to conduct a study on a larger dataset or more groups of the time budget in the future.

> **Finding 1**. Given a coverage criterion, we built 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. The multivariate lasso regression analysis shows that 13 metrics, such as API documentation, number of ancestors, the number of parents, and the number of setters, correlate with detectable faults.

4.2. RQ2. How many faults can be newly detected if one coverage criterion is applied after another?

Branch coverage is considered as an effective coverage criterion, which is widely-used to measure both manual and automated test generation (Fraser and Arcuri, 2012; Just et al., 2014;

Table 3
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	0.726	0.709	0.717	0.759	0.579	**0.970**	0.724	0.532	**0.766**	0.843	**0.804**	**0.837**	0.579	0.530	0.553	0.717
Direct branch	264	478	0.668	0.755	0.708	0.739	0.545	**0.959**	0.695	0.538	**0.763**	0.823	**0.790**	**0.835**	0.579	0.496	0.535	0.735
Line	115	627	0.457	**0.487**	0.472	0.707	0.667	0.035	0.071	0.514	**0.683**	0.448	**0.541**	**0.820**	0.318	0.122	0.176	0.650
Exception	251	491	0.664	0.711	0.689	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	**0.754**	0.633	0.779	0.556	0.031	0.059	0.509	**0.740**	0.618	**0.672**	**0.853**	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 4
Precision (**P**), recall (**R**), F-measure (**F**), and AUC of predicting detected faults for **10-minute** experiments.

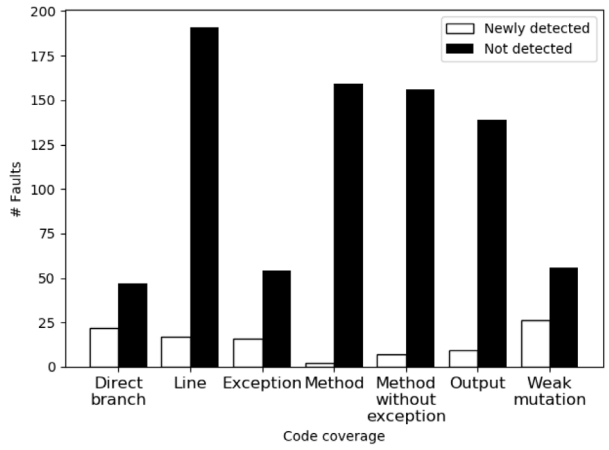
Coverage	# Faults		BayesNet + SMOTE				SVM + SMOTE				RandomForest+SMOTE				RandomForest			
	Detected	Undetected	P	R	F	AUC	P	R	F	AUC	P	R	F	AUC	P	R	F	AUC
Branch	294	448	0.716	0.735	0.724	0.759	0.586	**0.978**	0.733	0.536	**0.748**	0.847	**0.793**	**0.812**	0.588	0.568	0.578	0.716
Direct branch	290	452	0.728	0.720	0.723	0.764	0.578	**0.978**	0.726	0.531	**0.748**	0.828	**0.779**	**0.818**	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**	**0.810**	0.342	0.131	0.190	0.603
Exception	272	470	0.679	0.701	0.690	0.743	0.555	**0.966**	0.705	0.536	**0.752**	0.825	**0.787**	**0.821**	0.549	0.478	0.511	0.697
Method	136	606	0.493	**0.665**	0.567	0.751	0.500	0.026	0.051	0.507	**0.711**	0.596	**0.648**	**0.849**	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	**0.723**	0.591	**0.649**	**0.850**	0.463	0.226	0.304	0.718
Output	166	576	0.548	**0.712**	0.616	0.771	0.353	0.018	0.034	0.501	**0.709**	0.593	**0.647**	**0.829**	0.457	0.259	0.331	0.709
Weak mutation	281	461	0.689	0.694	0.693	0.745	0.562	**0.956**	0.708	0.524	**0.755**	0.813	**0.782**	**0.814**	0.554	0.495	0.523	0.707

Table 5
Top-5 metrics with the highest correlation of fault detection based on multivariate lasso regression analysis. A light-gray cell shows that the metric appears twice or more among eight coverage criteria.

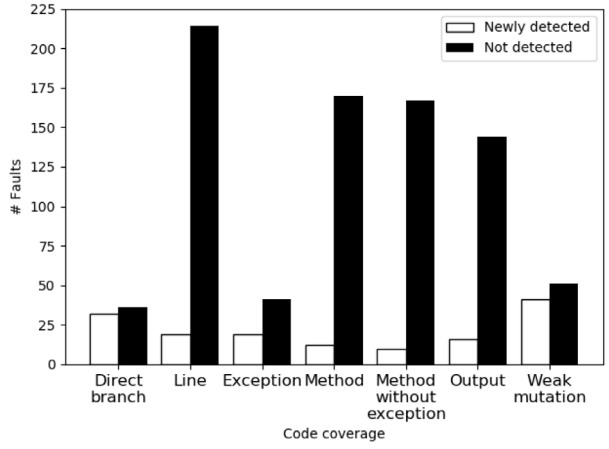
Coverage		Top-5 correlated metrics				
		The 1st metric	The 2nd metric	The 3rd metric	The 4th metric	The 5th metric
Group of **2-minute** experiments						
Branch	Metric	API documentation	Number of parents	Number of ancestors	Number of local setters	Depth of inheritance tree
	Coefficient	0.0883	-0.0532	0.0183	-0.0136	0.0131
Direct branch	Metric	API documentation	Number of parents	Number of ancestors	Number of local setters	Total number of local setters
	Coefficient	0.0765	-0.0338	0.0248	-0.0239	0.0174
Exception	Metric	Total number of local setters	Number of children	Number of setters	Number of ancestors	Number of local public attributes
	Coefficient	0.0092	-0.0069	-0.0067	0.0061	-0.0061
Line	Metric	Number of parents	Number of ancestors	API documentation	Number of local setters	Number of children
	Coefficient	-0.0510	0.0429	0.0355	-0.0144	-0.0132
Method	Metric	Number of ancestors	Number of setters	Clone classes	Coupling between object classes	Number of local public attributes
	Coefficient	0.0155	-0.0099	0.0092	-0.0082	-0.0076
Method no exception	Metric	Depth of inheritance tree	Number of setters	Coupling between object classes	Clone classes	Total number of setters
	Coefficient	0.0273	-0.0117	-0.0097	0.0088	0.0072
Output	Metric	Number of local setters	Total number of local setters	Coupling between object classes	Number of setters	Number of getters
	Coefficient	-0.0144	0.0134	-0.0106	-0.0069	0.0056
Weak mutation	Metric	API documentation	Number of parents	Number of ancestors	Number of local setters	Nesting level
	Coefficient	0.0451	-0.0293	0.0232	-0.0204	0.0129
Group of **10-minute** experiments						
Branch	Metric	API documentation	Number of parents	Depth of inheritance tree	Number of local setters	Clone classes
	Coefficient	0.0877	-0.0326	0.0244	-0.0113	0.0113
Direct branch	Metric	API documentation	Number of parents	Number of ancestors	Number of local public attributes	Number of local attributes
	Coefficient	0.0540	-0.0417	0.0198	-0.0110	0.0099
Exception	Metric	Total number of local setters	Number of ancestors	Nesting level	Number of local public attributes	Number of local methods
	Coefficient	0.0087	0.0058	0.0055	-0.0047	0.0032
Line	Metric	API documentation	Depth of inheritance tree	Number of parents	Number of local setters	Clone classes
	Coefficient	0.038	0.0331	-0.0316	-0.0218	0.0138
Method	Metric	Number of ancestors	Number of setters	Coupling between object classes	Nesting level else-if	Total number of setters
	Coefficient	0.0173	-0.0122	-0.0111	-0.0082	0.0070
Method no exception	Metric	Depth of inheritance tree	Clone classes	Coupling between object classes	Number of setters	Number of local setters
	Coefficient	0.0403	0.0117	-0.0105	-0.0103	-0.0084
Output	Metric	Clone classes	Number of setters	Coupling between object classes	Clone instances	Total number of local getters
	Coefficient	0.0114	-0.0101	-0.0075	-0.0074	-0.0069
Weak mutation	Metric	API documentation	Number of ancestors	Number of local setters	Coupling between object classes	Nesting level
	Coefficient	0.0823	0.0305	-0.0185	-0.0121	0.0094

Almasi et al., 2017). In Section 4.1, results in Tables 3 and 4 show that the branch coverage can detect the most number of faults among eight coverage criteria 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 the difference between the branch coverage and the other coverage criteria: for faults that cannot be detected by the branch coverage, how many faults can be newly detected by the other coverage criteria? We leverage this RQ to understand detectable faults with seven other different coverage criteria.

Fig. 3 shows the number of newly detected and not detected faults if a coverage criterion is applied after the branch coverage. Fig. 3(a) shows the number of newly detected and not detected faults by seven other coverage criteria for 2-minute experiments, compared to 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 criteria, the weak mutation coverage and the direct branch coverage should be the choices while using method coverage as a 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



(a) For 2-minute experiment



(b) For 10-minute experiment

Fig. 3. Number of newly detected and not detected faults, if a coverage criterion is applied after the branch coverage.

coverage, the exception coverage, and the weak mutation coverage cannot detect 47, 54, and 56 faults, respectively, compared to 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. 3(b) 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 is 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.

In Fig. 3, we showed that seven coverage criteria can be used to detect faults that are missed by the branch coverage. Meanwhile, in RQ1 (Section 4.1), we showed that each faulty file can be 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, if a coverage criterion is applied after the branch coverage. For each code coverage (except the branch coverage), we employed multivariate lasso regression analysis between each of 60 metrics and the flag of new fault detection for seven coverage criteria. Similar to the correlation analysis in RQ1(Section 4.1), we sorted all correlation coefficients in a descending order of the absolute values. Table 6 presents the names of top-5 metrics for each coverage criterion.

Table 6 contains two groups of experiments: 2 min of test generation per fault and 10 min per fault. In each group, we marked metrics that appear for two or more times in light-gray. We observed that in the 2-minute experiments, 9 metrics appear two or more times, including the number of local getters, the number of descendants, the number of setters, etc. Metrics of the number of descendants and the number of setters appear 5 times in total in seven coverage criteria. In the 10-minute experiments, 7 metrics appear two or more times, including the number of getters, the number of local public attributes, the number of descendants, etc. The metric of the number of local public attributes appears in each coverage criterion, and the metric of the number of getters appears 6 times in seven coverage criteria. The intersection of metrics in two groups consists of six metrics: the number of local getters, the number of descendants, the number of setters, the number of local setters, total number of local methods, and the clone classes.

Motivated by the newly detected faults between the branch coverage and other coverage criteria, we summarized the numbers of newly detected faults for all pairs of coverage criteria. Table 7 presents the number of newly detected faults if one coverage criterion is applied after another. A cell at the ith row and the jth column, called $\delta_{i,j}$, denotes the number of newly detected faults by applying the jth coverage after applying the ith coverage. Meanwhile, $\delta_{i,j}$ is equivalent to the number of not detected faults if applying the ith coverage after applying the jth coverage.

As shown in Table 7, we can choose a second coverage criterion if a coverage criterion is already applied in test generation. For the second choice, four coverage criteria, such as branch coverage, direct branch coverage, line coverage, and weak mutation coverage, can add more faults than the other four coverage criteria. From cells in each row, the branch coverage can add the most number of newly detected faults in all seven rows of the group of two-minute experiments. The branch coverage can add the most number of newly detected faults in four out of seven rows of the group of ten-minute experiments. This result suggests that applying one coverage criterion after another can add the number of newly detected faults.

Discussion. 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 the practice of using automated test generation, a user can set multiple coverage criteria to the automatic tool to enhance the ability of fault detection via branch coverage. We analyzed the correlation coefficient in this study with multivariate lasso regression analysis. In this study, we did not show the correlation between the code coverage and a defect class, i.e., the type of faults. A study on defect classes may help understand the role of coverage criteria in automated test generation.

Finding 2. Among all coverage criteria 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. The analysis on correlations suggests that six metrics, including the number of local getters and the number of descendants, appear in the top-5 correlations in both two groups of experiments.

Table 6
Top-5 metrics with the highest correlation with newly detected faults based on multivariate lasso regression analysis. A light-gray cell shows that the metric appears twice or more among seven coverage criteria.

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	Metric	Number of local getters	Nesting level	Number of descendants	Number of attributes	Clone classes
	Coefficient	0.0025	0.0014	0.0011	0.0011	-0.0009
Exception	Metric	Total number of local setters	Number of ancestors	Number of setters	Number of methods	Total number of getters
	Coefficient	0.0055	-0.0045	-0.0019	0.0016	-0.0015
Line	Metric	Lack of cohesion in methods 5	Number of local getters	Number of descendants	Number of local setters	Number of ancestors
	Coefficient	0.0026	0.0025	0.0013	0.0002	0.0011
Method	Metric	Total number of setters	Number of setters	Total number of methods	Total number of public methods	Number of local attributes
	Coefficient	0.0019	-0.0015	0.0010	-0.0009	0.0008
Method no exception	Metric	Number of descendants	Total number of setters	Number of setters	Number of local methods	Total number of public methods
	Coefficient	0.0021	0.0016	-0.0013	0.0010	-0.0009
Output	Metric	Number of descendants	Coupling between object classes	Number of local attributes	Lack of cohesion in methods 5	Number of setters
	Coefficient	0.0012	-0.0006	0.00049	0.00029	-0.0003
Weak mutation	Metric	Number of descendants	Nesting level else-if	Total number of local methods	Number of local methods	Number of setters
	Coefficient	0.0044	-0.0022	0.0016	-0.0016	0.0014
Group of **10-minute** experiments						
Direct branch	Metric	Number of descendants	Number of getters	Number of local getters	Number of local public attributes	Total number of public attributes
	Coefficient	0.0038	0.0037	-0.0033	-0.0032	0.0028
Exception	Metric	Number of descendants	Number of getters	Number of local public attributes	Clone classes	Total number of local methods
	Coefficient	0.0039	0.0027	-0.0025	-0.0021	-0.0016
Line	Metric	Number of public attributes	Number of local public attributes	Clone instances	Number of local setters	Total number of local public methods
	Coefficient	0.0069	-0.0068	0.0027	-0.0017	0.0010
Method	Metric	Number of setters	Number of getters	Total number of public attributes	Number of local public attributes	Total number of setters
	Coefficient	-0.0030	-0.0028	0.0029	0.0025	0.0021
Method no exception	Metric	Number of local public attributes	Number of setters	Number of getters	Total number of local public attributes	Total number of local getters
	Coefficient	-0.0036	-0.0033	0.0033	0.0024	-0.0017
Output	Metric	Number of local public attributes	Number of getters	Total number of public attributes	Total number of local methods	Total number of local attributes
	Coefficient	-0.0028	0.0028	0.0026	-0.0016	-0.0008
Weak mutation	Metric	Number of local public attributes	Total number of public attributes	Number of getters	Clone instances	Total number of local methods
	Coefficient	-0.0098	0.0093	0.0039	0.0036	-0.0030

Table 7
Number of newly detected faults if one coverage criterion is applied after another. A cell at the *i*th row and the *j*th column denotes the number of newly detected faults by applying the *j*th coverage after applying the *i*th coverage.

Coverage criterion	Branch	Direct branch	Exception	Line	Method	Method w/o exception	Output	Weak mutation
Group of two-minute experiments								
Branch	–	22	17	16	2	7	9	26
Direct branch	47	–	15	27	7	12	21	45
Exception	191	164	–	156	73	88	98	162
Line	54	40	20	–	12	13	23	50
Method	159	139	56	131	–	29	56	141
Method w/o exception	156	136	63	124	21	–	43	135
Output	139	126	54	115	29	24	–	125
Weak mutation	56	50	18	42	14	16	25	–
Group of ten-minute experiments								
Branch	–	32	19	19	12	10	16	41
Direct branch	36	–	14	37	12	9	18	49
Exception	214	205	–	194	92	100	111	199
Line	41	55	21	–	20	13	25	48
Method	170	166	55	156	–	25	62	155
Method w/o exception	167	162	62	148	24	–	53	156
Output	144	142	44	131	32	24	–	143
Weak mutation	54	58	17	39	10	12	28	–

4.3. RQ3. How many faults can be newly detected by increasing the time of test generation from two minutes to ten?

Intuitively, the more time we set to the test generation, the more faults we can detect. However, it is impractical if the test generation takes much time. This motivates us to explore how many new faults can be detected when we increase the time cost of test generation. In RQ3, we investigate the evolution of detected fault via changing the time cost of test generation per coverage criterion.

Given a fault, we can configure multiple coverage criteria in test generation to increase the number of detected faults. The cutoff time of each coverage criterion is called *time unit*. In the evaluation, we set two groups of experiments, including the time unit of two minutes and ten minutes. Table 8 shows all possible number of detected faults according to the number of time units. Each coverage criterion is only executed for one time unit. That is, the number of time units also equals to the number of coverage criteria in test generation.

As shown in Table 8, the number of detected faults increases with the number of time units. For a particular number of time units, the number of detected faults varies. For example, in the group of the two-minute experiment, the number of detected faults ranges from 196 to 332 if three time units (i.e., three coverage criteria) are set. This suggests that tests should choose coverage criteria to detect the most number of faults in a limited time.

For each coverage criterion of test generation, we showed the number of detected faults when we change the time unit per fault from 2 min to 10 min in Fig. 4. The increment of the time unit from 2 min to 10 min can add newly detected faults: the direct branch coverage, the weak mutation coverage, the line coverage, and the branch coverage can detect 51, 51, 43, and 32 new faults, respectively; the exception coverage, the method coverage, the method without exception coverage, and the weak mutation coverage can newly detect 16, 21, 19, and 25 faults, respectively. This result shows that when we aim to detect as many faults as possible, we can consider increasing the time cost of test generation. However, the choice of coverage criteria leads to a different number of detected faults.

Discussion. In RQ3, we showed that a fault may be not detected when we set the time cost of test generation to a short

Table 8
Minimum, median, and maximum number of detected faults by counting the time units. A time unit denotes two minutes and ten minutes in two groups of experiments, respectively.

Time unit		1 unit	2 units	3 units	4 units	5 units	6 units	7 units	8 units
Group of two minutes	Min	115	161	196	234	294	320	335	352
	Median	205	276.5	305	318	327.5	337	346	352
	Max	289	315	332	341	347	350	352	352
Group of ten minutes	Min	99	161	207	242	323	347	360	369
	Median	219	300.5	323	340	351	358.5	362.5	369
	Max	294	339	349	357	363	366	368	369

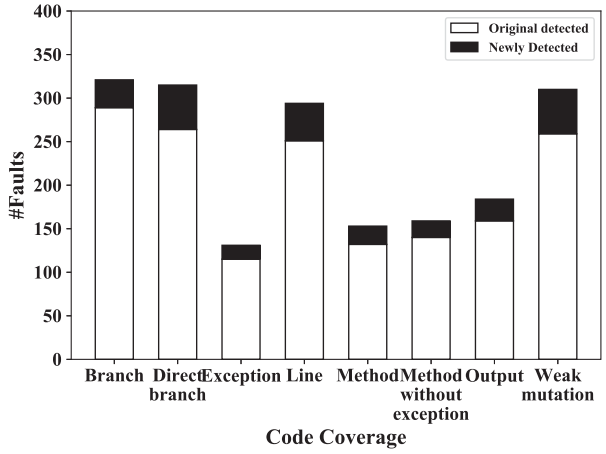


Fig. 4. Number of newly detected faults by eight coverage criteria from 2-minute to 10-minute experiments.

time, but can be detected when we increase the time cost (via adding other coverage criteria). In practice, a basic choice of code coverage in automated test generation can be the weak mutation coverage, the direct branch coverage, the line coverage, and the branch coverage.

> **Finding 3**. Among all code coverage under evaluation, the weak mutation coverage and the direct branch coverage can detect the most number of new faults when we increase the time cost of test generation; meanwhile, four coverage criteria can detect fewer new faults than the above two coverage criteria when we increase the time cost of test generation.

4.4. RQ4. Can we configure coverage criteria via learning from sampled faults?

The time cost of automated test generation cannot be ignored. In the dataset of this study (in Section 3.2), EvoSuite uses two minutes (or ten minutes) to generate tests for each faulty file and each code coverage. Taking the 2-minute experiments, i.e., data in Table 3, 8.24 days ($2 \times 8 \times 742 = 11872$ minutes) are spent in test generation. In practice, the assigned time budget of automated test generation is limited. In RQ4, we explore how to configure coverage criteria to minimize the number of undetected faults in a limited time budget.

Application. We aim to learn the configuration of coverage criteria from sampled faults and apply the learned configuration (the choice of coverage criteria) to generate tests for new faults. In a limited time budget, the learned configuration can be used to save the time cost of trying all configurations and avoid many undetected faults.

Problem definition. In this study, we considered the number of undetected faults and the time cost as two objectives for multiple times of test generation. We viewed the choice of eight coverage criteria as a solution to the optimization; each coverage criterion has three candidate values, i.e., not used, used for 2 min per fault, or used for 10 min per fault.[5] Therefore, the problem of choosing coverage is converted into a problem of multi-objective optimization, which minimizes the number of undetected faults and the time cost of test generation per fault. We employed existing algorithms of multi-objective optimization to illustrate the solutions.

Given a set S of coverage criteria, for each coverage criterion $s_i \in S$ $(1 \leq i \leq m)$, let an integer $c_i = k$ be the time budget of k minutes per fault of using s_i in test generation. In this study, c_i has three values: 0, 2, and 10 min, i.e., not used, used for 2 min per fault, and used for 10 min per fault. Given a set R of all faults, we define $R(s_i, c_i) \subseteq R$ as the detected faults by the coverage s_i with the cost c_i. Thus, a solution, i.e., the time cost of each coverage criterion, can be defined as $X = \{c_1, c_2, \ldots, c_m\}$.

Two objectives. We defined two objectives in our study: $f(X)$ for the number of undetected faults (comparing with the maximum number of detected faults) and $g(X)$ for the time cost per fault.

$$\text{Objective}_1\text{: minimize } f(X) = n_R - \bigcup_{s_i \in S} R(s_i, c_i)$$

$$\text{Objective}_2\text{: minimize } g(X) = \sum_{s_i \in S} c_i$$

where n_R is the number of all detected faults.

This work is to find out a solution X (i.e., the time of each coverage criterion) to satisfy the multi-objective optimization problem from sampled faults. This solution can be used to configure coverage criteria for new faults. Note that we do not aim to provide a resolution to optimize the choice of multiple coverage criteria. Instead, with the support of existing tests, we expect to show how the coverage criteria are chosen to minimize the number of undetected faults and the time cost.

Method. Fig. 5 presents an overview of our method based on multi-objective optimization. This method consists of two phases: the training phase and the testing phase. A solution to multi-objective optimization is a choice of multiple coverage criteria. In the training phase, the solution is learned from sampled faults and serves as the configuration of coverage criteria for new faults. In the test phase, the learned choice of coverage criteria is applied to new faults and used to generate tests.

We conducted 5 times 5-fold cross validation in the evaluation in two categories. In one category, the training set consists of one fold; in the other category, the training set consists of four folds. The different training sets are designed to show the influence of the size of the training set. In a test set, the optimal number of detected faults of each time cost is labeled. In each round, the number of non-detected faults $h(X, t)$ is recorded as follows,

$$h(X, t) = opt(t) - \bigcup_{s_i \in S} R(s_i, c_i)$$

[5] We limited the time of test generation to 2 min and 10 min because the available dataset (Salahirad et al., 2019) only provides the data of such time cost.

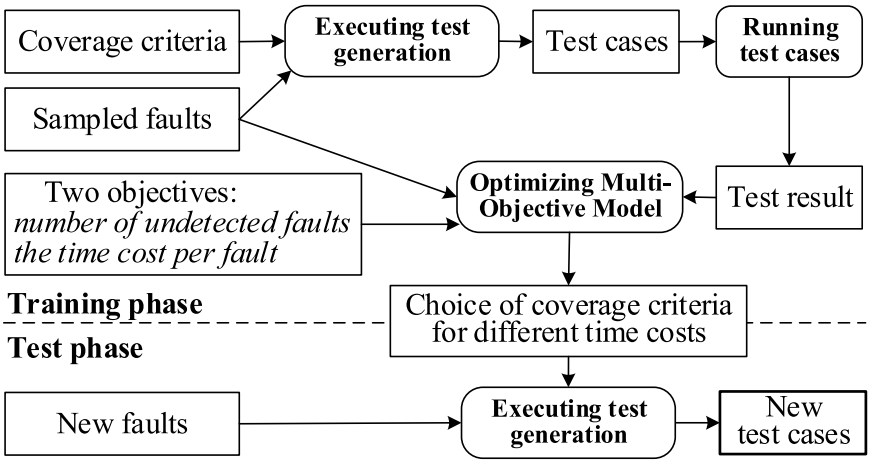


Fig. 5. Overview of our proposed method based on multi-objective optimization that learns a choice of multiple coverage criteria from sampled faults.

where t is the time cost of test generation per fault and $opt(t)$ is the maximum number of detected faults for t. The average of $h(X, t)$ of 25 rounds is reported as the final result. Note that the maximum number of detected faults $opt(t)$ is a function of the time cost t; that is, $opt(t)$ changes based on t.

Results. Fig. 6 shows the result by five algorithms of multi-objective optimization, including NSGA-II, DBEA, eMOEA, NSGA-III, and IBEA. In Fig. 6(a), we set one fold as the training set and the other four folds as the test set. When we compared to the maximum number of detected faults, five algorithms lose 3 to 13 faults on average. In general, the number of non-detected faults, i.e., $h(X, t)$, increases with the time cost of test generation per fault. Among five algorithms, NSGA-II, NSGA-III, DBEA, and eMOEA perform better than IBEA. eMOEA is better than NSGA-II when the time cost varies from 12 to 26 min or from 40 to 52 min. In Fig. 6(b), we set four folds as the training set and one fold as the test set. Five algorithms lose 0 to 1 faults on average. The number of non-detected faults $h(X, t)$ increases with the time cost of test generation per fault. Note that according to the definition of the objectives, $h(X, t)$ changes when the time cost t changes. This result suggests that if the time cost increase, the difference between the number of detected faults by the learned model and the optimal number of detected faults within the same time cost increases.

As shown in Fig. 6, the configuration of coverage criteria can be trained from sampled faults. This trained model can be directly applied to new faults. This suggests it is possible to learn such configurations from sampled faults. In practice, the training set can be set with a small number of sampled faults, such as a training set with one fold, i.e., 148 ($= 742/5$) faults.

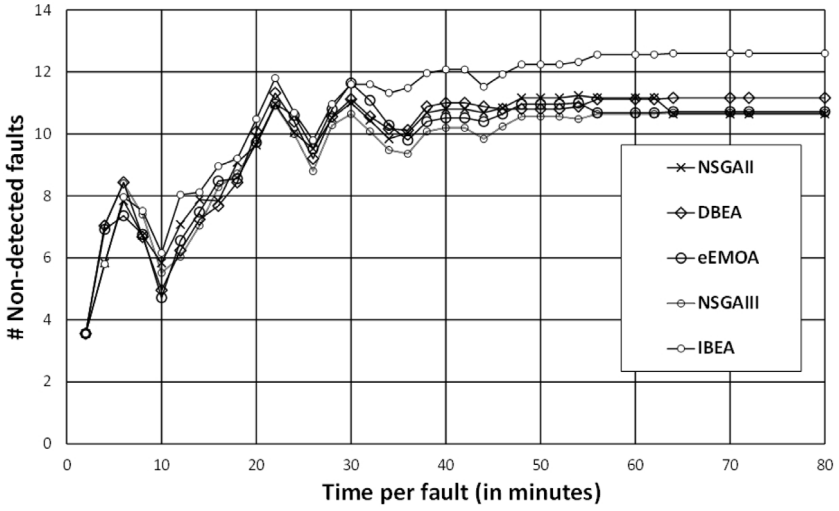
We explored the correlation between two objectives: the time cost and the number of undetected faults with five algorithms based on Spearman's rank correlation coefficient (Walpole et al., 2007; Xu et al., 2020). The Spearman's rank correlation coefficient is a measurement of the correlation between the ranking values of two independent variables. A higher absolute value of the Spearman's rank correlation coefficient indicates a higher correlation if the p-value suggests the statistical significance ($p < 0.01$ in this study). Table 9 shows the correlation coefficients and the p-values between two objectives for each algorithm. We found that all the correlation coefficients are over 0.7 with p-values lower than 0.01. Therefore, we can conclude that for each algorithm in the study, there is a high correlation between two objectives. That is, if the time cost of test generation increases, the number of undetected faults (compared with the maximum number of detected faults by all configurations of coverage criteria) increases. Such correlations are reasonable: the complexity of the combinations of coverage criteria increases with the number of candidate coverage criteria. Developers can apply the learned configuration of coverage criteria according to the setting of the time cost.

Table 9
Correlation between the time cost and the number of undetected faults based on Spearman's rank correlation coefficient.

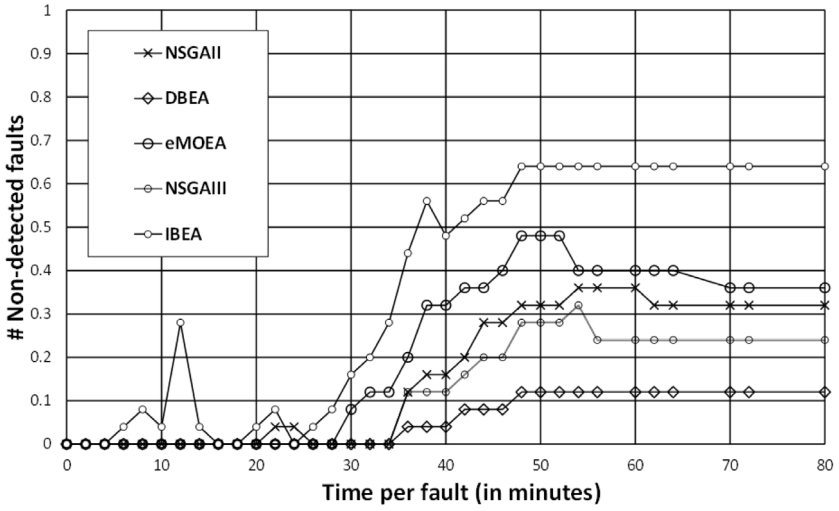
Algorithm	Coefficient	p-value
1 fold as the training set		
NSGAII	0.7818	4.81E−08
DBEA	0.8397	5.41E−10
eEMOA	0.6995	4.17E−06
IBEA	0.9731	5.80E−22
NSGAIII	0.8412	4.67E−10
4 folds as the training set		
NSGAII	0.8950	9.33E−13
DBEA	0.9168	2.67E−14
eEMOA	0.8781	8.99E−12
IBEA	0.9199	1.47E−14
NSGAIII	0.8797	7.39E−12

To further understand the results among five optimization algorithms, we conducted the one-tailed Wilcoxon signed-rank test to measure the difference of undetected faults between each pair of two algorithms. The Wilcoxon signed-rank test is a paired statistical test based on the ranks of values (Walpole et al., 2007). Table 10 shows the p-values in the Wilcoxon signed-rank test. We consider $p < 0.01$ as a statistical difference. That is, a cell of $p < 0.01$ shows that the number of undetected faults by the algorithm in its row is higher than that by the algorithm in its column. As shown in Table 10, in the category of 1 fold as the training set, the result by IBEA is higher than all the other four algorithms while the result by NSGA-III is lower than the other algorithms. In the category of 4 folds as the training set, DBEA achieves the lower number of undetected faults. This statistical test is consistent with the result of multi-objective optimization in Fig. 6.

Discussion. In our study, eight coverage criteria are used for the choice of code coverage. The volume of the search space of all solutions is 6561 (i.e., 3^8, where each coverage relates to three values and there are eight coverage criteria in total); this search space can be even exhaustively searched. However, our work used multi-objective optimization to illustrate the result.



(a) One out of five folds as the training set



(b) Four out of five folds as the training set

Fig. 6. Average number of non-detected faults $h(X, t)$ via training from sampled faults and testing on the other faults, where X is the solution from the training data and t is the time cost per fault. The average result is calculated based on 5-time 5-fold cross validation. Multi-objective optimization algorithms are used in the learning model, including NSGA-II, DBEA, eMOEA, NSGA-III, and IBEA.

Table 10
The p-values in the one-tailed Wilcoxon signed-rank test between each pair of two algorithms.

One-tailed (>)	NSGAII	DBEA	eEMOA	IBEA	NSGAIII
Category of **1 fold** as the training set					
NSGAII	–	0.7683	0.1562	1.0000	0.0011
DBEA	0.2372	–	0.0566	1.0000	0.0003
eEMOA	0.8485	0.9454	–	1.0000	0.0013
IBEA	3.03E−07	2.11E−06	1.18E−06	–	1.05E−06
NSGAIII	0.9989	0.9997	0.9988	1.0000	–
Category of **4 folds** as the training set					
NSGAII	–	5.43E−05	1.0000	1.0000	9.43E−05
DBEA	1.0000	–	1.0000	1.0000	0.9999
eEMOA	7.08E−05	4.36E−05	–	1.0000	4.53E−05
IBEA	1.15E−06	1.09E−06	1.25E−06	–	1.26E−06
NSGAIII	0.9999	0.0001	1.0000	1.0000	–

In other scenarios with more coverage criteria, multi-objective optimization should work better.

Finding 4. We converted the problem of choosing coverage into a problem of multi-objective optimization, which minimizes the number of undetected faults and the time cost of test generation per fault. The solution to this problem can be learned from sampled faults and directly applied to new faults.

5. Threats to validity

Threats to construct validity. Our study used the dataset by Salahirad et al. (2019) to study the ability of fault detection by automated test generation with different coverage criteria. Collecting data of test execution is time-consuming. This dataset contains the test results with eight coverage criteria 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 4.2, we answered RQ2 via showing newly detected faults and missed faults between different coverage criteria. 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 4.1, 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 criteria. A large scale evaluation could explain the benefit of predicting fault detection. We note that RQ4 is not designed to find out the best choice; RQ4 is designed to present the choices of coverage criteria with a limited time budget of test generation. In Section 4.4, we illustrated the multi-objective optimization of the number of undetected faults and the time cost in test generation per fault. In the evaluation, we used available data of our dataset to conduct an optimization instance of eight coverage criteria. This evaluation can be enhanced via increasing the number of faults.

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 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 criteria.

6. Related work

We present the related work in two categories: the study on test generation and the predictive tasks in testing and debugging.

6.1. Study on test generation

Automated test generation has been widely-studied. Many automated tools are proposed, including Randoop by Pacheco et al. (2007), Java PathFinder by Anand et al. (2007), and EvoSuite by Fraser and Arcuri (2011). For example, EvoSuite is used to detect faults or generate specific tests due to its high usability. Fraser and Arcuri (2015) 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. (2014) conducted an empirical study on the test adequacy in 300 open-source projects. Just et al. (2014) studied the usefulness of mutation coverage in detecting real faults. Wu et al. (2020) analyzed the influence on invalid bug reports in software aging. Zhang et al. (2019a) compared and analyzed the issue reports in GitHub in two major platforms, desktop software and mobile Apps. Najafi et al. (2019) conducted an industrial study on the test effectiveness via historical test executions.

In the study of test generation, Salahirad et al. (2019) 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. (2019). Different from the existing work, our study shows that it is feasible to identify detectable faults with given coverage criteria; 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.

6.2. Predictive tasks in testing and debugging

Automated tools of testing as well as debugging can reduce the manual effort by developers. Existing studies have explored the reliability and effectiveness of automated tools for practical tasks. Le and Lo (2013), Le et al. (2015b) designed a predictive method on the effectiveness of fault localization, which ranks suspicious source code to assist debugging. Le et al. (2015a) proposed to identify the feasibility of applying automated program repair techniques to generate patches. Jiang et al. (2017) proposed to identify the cause of test alarms in the integration testing of real systems. Grano et al. (2019) developed machine-learning methods to predict the coverage by tests. Xu et al. (2019) characterized higher-order functions in Scala language to prioritize functions that should be tested. Le et al. (2017) designed a method to identify the reliability of bug localization, which recommends related source files to a given bug report. Li et al. (2018) proposed a recommendation method to rank exception handling strategies for potential exceptions in source code. Gu et al. (2019) designed a predictive method for crashing fault residence to identify whether the root cause of a crash resides in the stack trace. Zhang et al. (2019b) proposed a similarity-based approach to generate tags for unlabeled bug reports. Zhou et al. (2020) used deep learning to recognize bug-specific named entities in software artifacts.

Huang et al. (2019) proposed a refined method CBS+ for the effort-aware just-in-time defect prediction and investigated existing techniques of supervised and unsupervised defect prediction. Yan et al. (2020) proposed a framework integrating just-in-time prediction and positioning. Their work extracts features of 14 code change levels and predicts whether there are potential errors of the changes via the logistic regression. Xia et al. (2016) proposed a method that predicts whether a bug report is indeed modified or redistributed. This method treats each part of a bug report as a tag and then uses multi-tag learning to predict new bug reports. Thung et al. (2015) proposed a method for defect report classification, which combines active learning and semi-supervised learning to effectively reduce the number of training samples. Tantithamthavorn and Hassan (2018) have conducted an experience report on the pitfalls and the opportunities for practical defect prediction. Jiarpakdee et al. (2020) presented the impact of correlation between metrics in the predictive models on defect prediction.

7. Conclusions

We conducted a 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 fault. Experimental results show that RandomForest with SMOTE can achieve effective prediction on detected faults for all coverage criteria; one coverage criterion can be used to supplement another coverage criteria and add the number of detected faults; in a limited time budget of test generation, trying multiple coverage criteria leads to more detected faults than using one single type of code coverage. Our experiment also suggests it is possible to learn a configuration of coverage criteria from sampled faults and then apply the configuration to detect new faults.

In future work, we plan to explore the reason for 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 manually analyze the metrics of faulty code to further understand the potential dependency among metrics, faults, and tests.

CRediT authorship contribution statement

Ping Ma: Data curation, Software, Validation, Visualization, Writing - original draft, Writing - review & editing. **Hangyuan Cheng:** Validation. **Jingxuan Zhang:** Writing - review & editing. **Jifeng Xuan:** Conceptualization, Data curation, Funding acquisition, Methodology, Project administration, Software, Writing - original draft.

Acknowledgments

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 and the National Natural Science Foundation of China under Grant Nos. 61872273 and 61902181.

References

Almasi, M.M., Hemmati, H., Fraser, G., Arcuri, A., 2017. 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. IEEE, pp. 263–272.

Alshahwan, N., Harman, M., 2014. 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. pp. 181–192. http://dx.doi.org/10.1145/2610384.2610413.

Anand, S., Pasareanu, C.S., Visser, W., 2007. 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. pp. 134–138. http://dx.doi.org/10.1007/978-3-540-71209-1_12.

Asafuddoula, M., Ray, T., Sarker, R.A., 2015. A decomposition-based evolutionary algorithm for many objective optimization. IEEE Trans. Evol. Comput. 19 (3), 445–460. http://dx.doi.org/10.1109/TEVC.2014.2339823.

Blondeau, V., Etien, A., Anquetil, N., Cresson, S., Croisy, P., Ducasse, S., 2017. 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. pp. 58–68. http://dx.doi.org/10.1109/ICSME.2017.68.

Chawla, N.V., Bowyer, K.W., Hall, L.O., Kegelmeyer, W.P., 2002. SMOTE: synthetic minority over-sampling technique. J. Artificial Intelligence Res. 16, 321–357. http://dx.doi.org/10.1613/jair.953.

Chekam, T.T., Papadakis, M., Traon, Y.L., Harman, M., 2017. 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. pp. 597–608. http://dx.doi.org/10.1109/ICSE.2017.61.

Cheng, H., Ma, P., Zhang, J., Xuan, J., 2020. Can this fault be detected by automated test generation: A preliminary study. In: Proceedings of the 2nd International Workshop on Intelligent Bug Fixing, IBF 2020, London, Ontario, Canada, February 18-21, 2020. IEEE Computer Society.

Deb, K., Agrawal, S., Pratap, A., Meyarivan, T., 2002. A fast and elitist multiobjective genetic algorithm: NSGA-II. IEEE Trans. Evol. Comput. 6 (2), 182–197. http://dx.doi.org/10.1109/4235.996017.

Deb, K., Jain, H., 2014. An evolutionary many-objective optimization algorithm using reference-point-based nondominated sorting approach, part I: Solving problems with box constraints. IEEE Trans. Evol. Comput. 18 (4), 577–601. http://dx.doi.org/10.1109/TEVC.2013.2281535.

Fraser, G., Arcuri, A., 2011. 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. pp. 416–419. http://dx.doi.org/10.1145/2025113.2025179.

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

Fraser, G., Arcuri, A., 2014. A large scale evaluation of automated unit test generation using evosuite. ACM Trans. Softw. Eng. Methodol. 24 (2), 8.

Fraser, G., Arcuri, A., 2015. 1600 faults in 100 projects: automatically finding faults while achieving high coverage with evosuite. Empir. Softw. Eng. 20 (3), 611–639. http://dx.doi.org/10.1007/s10664-013-9288-2.

Gay, G., 2018. To call, or not to call: Contrasting direct and indirect branch coverage in test generation. In: Galeotti, J.P., Gorla, A. (Eds.), Proceedings of the 11th International Workshop on Search-Based Software Testing, ICSE 2018, Gothenburg, Sweden, May 28-29, 2018. ACM, pp. 43–50. http://dx.doi.org/10.1145/3194718.3194719.

Grano, G., Titov, T.V., Panichella, S., Gall, H.C., 2019. Branch coverage prediction in automated testing. J. Softw.: Evol. Process.

Gu, Y., Xuan, J., Zhang, H., Zhang, L., Fan, Q., Xie, X., Qian, T., 2019. Does the fault reside in a stack trace? Assisting crash localization by predicting crashing fault residence. J. Syst. Softw. 148, 88–104. http://dx.doi.org/10.1016/j.jss.2018.11.004.

Hall, M.A., Frank, E., Holmes, G., Pfahringer, B., Reutemann, P., Witten, I.H., 2009. The WEKA data mining software: An update. SIGKDD Explorations 11 (1), 10–18. http://dx.doi.org/10.1145/1656274.1656278.

Huang, Q., Xia, X., Lo, D., 2019. Revisiting supervised and unsupervised models for effort-aware just-in-time defect prediction. Empir. Softw. Eng. 24 (5), 2823–2862. http://dx.doi.org/10.1007/s10664-018-9661-2.

Jiang, H., Li, X., Yang, Z., Xuan, J., 2017. 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. pp. 712–723. http://dx.doi.org/10.1109/ICSE.2017.71.

Jiarpakdee, J., Tantithamthavorn, C., Hassan, A.E., 2020. The impact of correlated metrics on the interpretation of defect models. IEEE Trans. Softw. Eng..

Just, R., Jalali, D., Inozemtseva, L., Ernst, M.D., Holmes, R., Fraser, G., 2014. 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. pp. 654–665. http://dx.doi.org/10.1145/2635868.2635929.

Kochhar, P.S., Thung, F., Lo, D., Lawall, J.L., 2014. 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. pp. 215–222. http://dx.doi.org/10.1109/APSEC.2014.42.

Le, X.D., Le, T.B., Lo, D., 2015a. 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. pp. 427–437. http://dx.doi.org/10.1109/ISSRE.2015.7381836.

Le, T.B., Lo, D., 2013. 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. pp. 310–319. http://dx.doi.org/10.1109/ICSM.2013.42.

Le, T.B., Lo, D., Thung, F., 2015b. Should I follow this fault localization tool's output? - automated prediction of fault localization effectiveness. Empir. Softw. Eng. 20 (5), 1237–1274. http://dx.doi.org/10.1007/s10664-014-9349-1.

Le, T.B., Thung, F., Lo, D., 2017. Will this localization tool be effective for this bug? Mitigating the impact of unreliability of information retrieval based bug localization tools. Empir. Softw. Eng. 22 (4), 2237–2279. http://dx.doi.org/10.1007/s10664-016-9484-y.

Li, Y., Ying, S., Jia, X., Xu, Y., Zhao, L., Cheng, G., Wang, B., Xuan, J., 2018. 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. IEEE Computer Society, pp. 104–114. http://dx.doi.org/10.1109/ICECCS2018.2018.00019.

Martin, D.B., Rooksby, J., Rouncefield, M., Sommerville, I., 2007. '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. pp. 602–611. http://dx.doi.org/10.1109/ICSE.2007.1.

Najafi, A., Shang, W., Rigby, P.C., 2019. Improving test effectiveness using test executions history: An industrial experience report. In: Sharp, H., Whalen, M. (Eds.), Proceedings of the 41st International Conference on Software Engineering: Software Engineering in Practice, ICSE (SEIP) 2019, Montreal, QC, Canada, May 25-31, 2019. IEEE / ACM, pp. 213–222. http://dx.doi.org/10.1109/ICSE-SEIP.2019.00031.

Namin, A.S., Kakarla, S., 2011. 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. pp. 342–352. http://dx.doi.org/10.1145/2001420.2001461.

Pacheco, C., Lahiri, S.K., Ernst, M.D., Ball, T., 2007. Feedback-directed random test generation. In: 29th International Conference on Software Engineering (ICSE 2007), May 20-26, 2007. pp. 75–84. http://dx.doi.org/10.1109/ICSE.2007.37.

Rojas, J.M., Campos, J., Vivanti, M., Fraser, G., Arcuri, A., 2015. Combining multiple coverage criteria in search-based unit test generation. In: de Oliveira Barros, M., Labiche, Y. (Eds.), Search-Based Software Engineering - 7th International Symposium, SSBSE 2015, Bergamo, Italy, September 5-7, 2015, Proceedings. In: Lecture Notes in Computer Science, 9275, Springer, pp. 93–108. http://dx.doi.org/10.1007/978-3-319-22183-0_7.

Salahirad, A., Almulla, H., Gay, G., 2019. Choosing the fitness function for the job: Automated generation of test suites that detect real faults. Softw. Test. Verif. Reliab. 29 (4–5), (in press).

Staats, M., Gay, G., Heimdahl, M.P.E., 2012. 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. pp. 870–880. http://dx.doi.org/10.1109/ICSE.2012.6227132.

Sun, Y., Yen, G.G., Yi, Z., 2019. IGD indicator-based evolutionary algorithm for many-objective optimization problems. IEEE Trans. Evol. Comput. 23 (2), 173–187. http://dx.doi.org/10.1109/TEVC.2018.2791283.

Tantithamthavorn, C., Hassan, A.E., 2018. An experience report on defect modelling in practice: pitfalls and challenges. In: Paulisch, F., Bosch, J. (Eds.), Proceedings of the 40th International Conference on Software Engineering: Software Engineering in Practice, ICSE (SEIP) 2018, Gothenburg, Sweden, May 27 - June 03, 2018. ACM, pp. 286–295. http://dx.doi.org/10.1145/3183519.3183547.

Thung, F., Le, X.D., Lo, D., 2015. Active semi-supervised defect categorization. In: 2015 IEEE 23rd International Conference on Program Comprehension. pp. 60–70.

Walpole, R.E., Myers, S.L., Ye, K., Myers, R.H., 2007. Probability and statistics for engineers and scientists. Pearson.

Wu, X., Zheng, W., Pu, M., Chen, J., Mu, D., 2020. Invalid bug reports complicate the software aging situation. Softw. Qual. J. 28 (1), 195–220. http://dx.doi.org/10.1007/s11219-019-09481-2.

Xia, X., Lo, D., Shihab, E., Wang, X., 2016. Automated bug report field reassignment and refinement prediction. IEEE Trans. Reliab. 65 (3), 1094–1113.

Xu, Y., Jia, X., Xuan, J., 2019. 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. In: Internetware, http://dx.doi.org/10.1145/3361242.3361256.

Xu, Y., Wu, F., Jia, X., Li, L., Xuan, J., 2020. Mining the use of higher-order functions: An exploratory study on scala programs. Empir. Softw. Eng. http://dx.doi.org/10.1007/s10664-020-09842-7.

Xuan, J., Martinez, M., Demarco, F., Clement, M., Marcote, S.R.L., Durieux, T., Berre, D.L., Monperrus, M., 2017. Nopol: Automatic repair of conditional statement bugs in java programs. IEEE Trans. Softw. Eng. 43 (1), 34–55. http://dx.doi.org/10.1109/TSE.2016.2560811.

Xuan, J., Monperrus, M., 2014. 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. pp. 52–63. http://dx.doi.org/10.1145/2635868.2635906.

Yan, M., Xia, X., Fan, Y., Hassan, A.E., Lo, D., Li, S., 2020. Just-in-time defect identification and localization: A two-phase framework. IEEE Trans. Softw. Eng..

Yuan, Y., Xu, H., Wang, B., Yao, X., 2016. A new dominance relation-based evolutionary algorithm for many-objective optimization. IEEE Trans. Evol. Comput. 20 (1), 16–37. http://dx.doi.org/10.1109/TEVC.2015.2420112.

Zhang, T., Chen, J., Luo, X., Li, T., 2019a. Bug reports for desktop software and mobile apps in github: What's the difference?. IEEE Softw. 36 (1), 63–71. http://dx.doi.org/10.1109/MS.2017.377142400.

Zhang, T., Li, H., Xu, Z., Liu, J., Huang, R., Shen, Y., 2019b. Labelling issue reports in mobile apps. IET Softw. 13 (6), 528–542. http://dx.doi.org/10.1049/iet-sen.2018.5420.

Zhang, F., Zheng, Q., Zou, Y., Hassan, A.E., 2016. 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. pp. 309–320. http://dx.doi.org/10.1145/2884781.2884839.

Zhou, C., Li, B., Sun, X., 2020. Improving software bug-specific named entity recognition with deep neural network. J. Syst. Softw. 165, 110572. http://dx.doi.org/10.1016/j.jss.2020.110572.

Zhu, H., Hall, P.A.V., May, J.H.R., 1997. Software unit test coverage and adequacy. ACM Comput. Surv. 29 (4), 366–427. http://dx.doi.org/10.1145/267580.267590.

Zimmermann, T., Nagappan, N., Gall, H.C., Giger, E., Murphy, B., 2009. 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. pp. 91–100. http://dx.doi.org/10.1145/1595696.1595713.