Mutation-based Test-Case Prioritization in Software Evolution

Yiling Lou, Dan Hao*, Lu Zhang
Key Laboratory of High Confidence Software Technologies (Peking University), MoE, China
Institute of Software, School of EECS, Peking University, China
Email: {yilinglou,haodan,zhanglucs}@pku.edu.cn

Abstract—**During software evolution, to assure the software quality, test cases for an early version tend to be reused by its latter versions. As a large number of test cases may aggregate during software evolution, it becomes necessary to schedule the execution order of test cases so that the faults in the latter version may be detected as early as possible, which is test-case prioritization in software evolution. In this paper, we proposed a novel test-case prioritization approach for software evolution, which first uses mutation faults on the difference between the early version and the latter version to simulate real faults occurred in software evolution, and then schedules the execution order of test cases based on their fault-detection capability, which is defined based on mutation faults. In particular, we present two models on calculating fault-detection capability, which are statistics-based model and probability-based model. Moreover, we conducted an experimental study and found that our approach with the statistics-based model outperforms our approach with the probability-based model and the total statement coverage-based approach, and slightly outperforms the additional statement-coverage based approach in many cases. Furthermore, compared with the total or additional statement coverage-based approach, our approach with either the statistics-based model or the probability-based model tends to be stably effective when the difference on the source code between the early version and the latter version is non-trivial.**

I. Introduction

During software evolution, programmers modify an early version of a project and get its latter version due to the following reasons, improving software quality, adding a new functionality, modifying existing functionalities, and so on [1], [2]. Although the early version may have been fully tested, it is still necessary to test the latter version because modification on the early version may induce faults. To test the latter version efficiently, it is natural to reuse the existing test cases designed for the early version so as to reduce the cost on test-case generation for the latter version.

On the other hand, in software evolution, the number of test cases grows rapidly. Let us take Time&Money[1] as an example. In software evolution, from Time&Money 0.2 to Time&Money 0.3, the number of test cases increases from 104 to 146 (about 40.38%). That is, due to the rapid growth on test cases, a project may have an extremely large set of test cases. However, in some testing scenarios, the testing resources (e.g., time and efforts) are limited, and thus it is necessary to schedule the execution order of test cases so as to detect faults as early as possible. That is, test-case prioritization is important in software evolution.

Although various test-case prioritization approaches have been proposed in the literature [4], [5], [6], [7], [8], [9], [10], [11], [12], [13], [14], [15], [16], these approaches may be not effective in software evolution. Typically, these approaches prioritize test cases mainly based on some structural coverage (e.g., statement coverage [17] or method coverage [18]) of an early version. However, in software evolution, the different source code between the early version and the latter version is usually non-trival. For example, from Time&Money 0.2 to Time&Money 0.3, the number of executable statements increases from 1,059 to 1,403. That is, in the evolution of Time&Money, about 50% of the code is added to an early version. Therefore, it may be not effective to prioritize test cases based on just the coverage information of an early version. Although some approaches [19], [13], [12], [20] prioritize test cases based on the coverage of change occurring in software evolution, such change is not necessarily related to faults occurring in software evolution, and thus these change based approaches may be not effective either.

Furthermore, the existing test-case prioritization approaches are mostly evaluated in an unreal testing scenario [18], [3], [21], [17], [7], [22], [23]. In particular, in their evaluation, the latter versions are usually not collected from practice, but manually constructed by adding faults in the early version [18], [3], [21]. Furthermore, their proposed approaches are evaluated based on detecting the same faults [18], [3], [21]. Although this testing scenario is far from real, it is widely used in test-case prioritization. As test-case prioritization has the preceding effectiveness and evaluation issues, in this paper, we plan to present a practical test-case prioritization approach for software evolution and evaluate the proposed approach in real software evolution scenarios.

Intuitively, test cases revealing faults in the latter version should be executed early. As reported by previous work [24], [25], mutation faults generated in mutation testing can serve as the substitute for real faults, and thus we may use mutation faults to simulate real faults in the latter version and then use mutation faults to guide test-case prioritization. Furthermore, the latter version is modified based on an early version, which tends to be fully tested. Therefore, the faults in the latter version mostly come from the modification from the early

*Corresponding author
[1]Time&Money is an open-source project, which is used in the evaluation of previous work [3] and available at http://timeandmoney.sourceforge.net.

version to the latter version.

Based on the preceding intuition, in this paper, we present a mutation-based test-case prioritization approach in software evolution. For any two versions of a project, which are denoted as P_0 and P_1, the set of test cases designed for P_0 and used to test P_1 is denoted as T. The proposed mutation-based test-case prioritization approach takes the source code of P_0 and P_1 as input, and outputs the prioritized test suite for P_1. The proposed approach first calculates the fault-detection capability of each test case based on its number of killing[2] mutants which occur on the difference between P_0 and P_1, and then prioritizes test cases in T based on their fault-detection capability. In particular, in the proposed approach, we present two models (i.e., statistics-based model and probability-based model) on calculating the fault-detection capability of test cases using their killing mutants. The statistics-based model calculates the fault-detection capability of a test case based on the number of mutants killed by the test case, whereas the probability-based model calculates the fault-detection capability of a test case considering the distribution of mutants.

To evaluate the effectiveness of the proposed approach, we conducted an experimental study on 14 versions of 3 open-source Java projects by comparing the total and additional coverage-based approaches, which are usually used as the control approaches in previous work [26], [3]. As the Java projects in the experimental study use JUnit testing framework, each JUnit test case has two levels of test cases: test-method level and test-class level. Therefore, our experimental study uses the two levels of test cases as two granularities of test cases. From the experimental results, when prioritizing test cases at the test-method level, our approach with the statistics-based model significantly outperforms our approach with the probability-based model and the total approach, and is slightly more effective than the additional approach without significant difference. Furthermore, our approach is more effective when prioritizing test cases at the test-method level than at the test-class level. Moreover, even if non-trivial changes occur from the early version to the latter version, our approach is still steady, especially compared with the total and additional coverage-based approaches.

The contributions of the paper are summarized as:

- A novel test-case prioritization approach for software evolution, which prioritizes test cases based on their killing mutants on the different statements between the early version and the latter version.
- An experimental study showing that the proposed approach is more effective than the total approach, and competitive compared with the additional approach.

[2]In mutation testing, mutants are generated by performing some mutation operators (e.g., deleting a statement and replacing a constant with another one) on the program. That is, a mutant can be viewed as a faulty program. For any test case t, if the original program behaves different from a mutant, this mutant is said to be killed by the test case t.

II. Approach

In software evolution, it is natural to reuse existing test cases of an early version for its latter version because the two versions belonging to the same project and their source codes have similarity. As the early version is usually tested, faults in the latter version mostly lie in the difference between the two versions. Furthermore, as mutants generated by mutation testing can be representative of real faults [24], [25], we may use mutation faults in the difference to simulate real faults in software evolution.

Based on the preceding intuition, we present a mutation-based test-case prioritization approach, which schedules the execution order of test cases based on their killing mutants that occur on the modification. P_0 represents an early version and P_1 represents its latter version. In the test suite designed for testing P_0, some test cases cannot be used in testing P_1 because compiling errors may occur when running the test cases on P_1. By removing these test cases, we get the set of test cases that are designed for P_0 and can be used to test P_1, which is denoted as T. The mutation-based test-case prioritization approach proposed in this paper aims to schedule the execution order of test cases in T to test P_1.

In particular, the proposed approach consists of the following steps. First, our approach identifies modification on the original program (see Section II-A). Second, our approach generates mutants whose mutation operators occur on only the modification (see Section II-B). Third, our approach schedules the execution order of test cases based on their fault-detection capability (see Section II-C).

A. Modification Identification

For any two versions in software evolution P_0 and P_1, the proposed approach compares the source code of P_0 and P_1 by using some tools (e.g., the $diff$ command provided by Linux Operating System), and maps the difference on the source code of P_0. The difference between P_0 and P_1 can be classified into the following categories: (1) source code that lies in P_0, but not in P_1, (2) source code that lies in P_0 and P_1, but they are slightly different, and (3) source code that does not lie in P_0, but appears in P_1.

For ease of representation, we represent the modification in the granularity of statements, which can also be represented in other granularities (e.g., methods and classes, discussed in Section IV). Using the granularity of statements to represent modification, the preceding three types of modification can be represented by (1) deleting statements in P_0, (2) modifying statements in P_0, and (3) adding statements to P_0.

The problem of test-case prioritization is proposed for at least one purpose, efficiency. Therefore, it is important to take efficiency into consideration while designing a new test-case prioritization approach. As it is costly to generate and run mutants [27], [28], the proposed approach is designed to prioritize test cases based on the mutants of P_0 rather than P_1 so that in software evolution programmers can reuse the results of mutant execution on an early version. This also explains why the proposed approach maps the preceding modification to the source code of P_0 rather than that of P_1.

In particular, for some modification belonging to (1) or (2), the proposed approach labels the modification by the corresponding statement (i.e., the deleted statement in P_0 or the modified statement in P_0). For a modification belonging to (3), it is hard to label the modification by the corresponding statement in P_0 because the added statement does not exist in P_0. To label such modifications, currently the proposed approach labels the statement whose position in the program appears just before the added statement. For example, if the statement should be added to just after the statement s, the proposed approach labels this added statement by s[3]. However, this label is not precise, and we will discuss this in Section IV.

Through the preceding process, the proposed approach labels the difference between P_0 and P_1, which is a set of statements in P_0 and denoted as $\triangle P$.

B. Mutant Generation

As mutation faults can be representative of real faults in software testing [24], [25], a test case with high capability on detecting mutation faults may have high capability on detecting real faults [17]. Based on this intuition, we simulate real faults in software evolution by mutants. Furthermore, as the early version P_0 is usually tested, the faults in P_1 usually come from the difference between P_0 and P_1. Therefore, the proposed approach generates mutants whose mutation operators occur on only the statements in $\triangle P$.

In implementing the proposed approach, it is hard to use any existing mutation tool (e.g., Javalanche [29]) by specifying the statements where mutation operators occur. Therefore, in implementing we actually generate all the mutants for P_0 and then select the mutants whose mutation operators occur on the statements in $\triangle P$. Note that in the remaining of this paper, without explicit explanation, the mutants are referred to those whose mutation operators occur on the statements in $\triangle P$.

C. Execution-Order Scheduling

For each selected mutant, the proposed approach records whether it is killed by each test case within T, which is used to calculate the fault-detection capability. Then the proposed approach schedules the execution order of test cases based on the descendent order of their fault-detection capability.

In particular, we present two models on defining the fault-detection capability based on the mutants test cases kill. The first one is statistics-based model, which calculates the fault-detection capability of a test case based on the total number of mutants it kills. The second one is probability-based model, which calculates the fault-detection capability of a test case considering the distribution of mutants.

1) Statistics-based Model: Intuitively, the more mutants a test case kills, the higher probability a test case may have on detecting real faults in P_1. Therefore, we can use the total number of mutants a test case kills to define its fault-detection capability.

Supposed that through the process described in Section II-B there are totally N mutants, which are denoted by m_1, m_2, $\ldots$, m_N, and test suite T consists of M test cases, which are denoted by t_1, t_2, $\ldots$, t_M. We use a Boolean matrix K to represent whether a mutant is killed by a test case in T. That is, the value of any element in K (denoted as $K[i][j]$) represents whether a mutant m_i $(1 \leq i \leq N)$ is killed by a test case t_j $(1 \leq j \leq M)$. If the test case t_j kills the mutant m_i, $K[i][j]$ is 1; otherwise, $K[i][j]$ is 0.

Based on the matrix K, the fault-detection capability of any test case t_j, which is denoted as $weight(t_j)$, is defined as follows in the statistics-based model.

$$weight(t_j) = \sum_{i=1}^{N} K[i][j] \tag{1}$$

For any test case t_j, the more mutants it kills, the higher $weight(t_j)$ is, and the higher probability it has on detecting faults in P_1.

2) Probability-based Model: As faults in P_1 mostly lie in the modified statements between P_0 and P_1 (i.e., $\triangle P$), the fault-detection capability of a test case can be calculated based on its capability in detecting mutation faults in each statement in $\triangle P$.

The mutants generated for $\triangle P$ are not evenly distributed in statements. For example, in JAVA programs, the statement "if(x==y)" tends to have more mutants (e.g., "if(x<=y)", "if(x>=y)") than the statement "system.out.println("error!")". That is, some types of statements have many mutants, whereas some do not. Considering the uneven distribution of mutants, the proposed approach first groups the mutants whose mutation operators occur on the same statement into one group and then estimates how likely a test case t_j detects the faults in one statement (denoted as s_i) based on the following two numbers: (1) the number of mutants whose mutation operators occur on this statement (denoted as $Num_t(i,j)$), and (2) the number of mutants whose mutation operators occur on this statement and that are killed by t_j (denoted as $Num_k(i,j)$). That is, how likely t_j detects faults in s_i, which is denoted as $P(i,j)$, is defined as follows.

$$P(i,j) = \frac{Num_k(i,j)}{Num_t(i,j)} \tag{2}$$

Then the probability-based model calculates the fault-detection capability of a test case based on how likely this test case detects faults in all the statements of $\triangle P$. Supposed that the set of statements in $\triangle P$ can be represented by $\triangle P$=$\{s_1, s_2, \ldots, s_D\}$, the probability-based model defines the fault-detection capability of any test case t_j according to the following equation.

$$weight(t_j) = \sum_{i=1}^{D} P(i,j)/D \tag{3}$$

For any test case, its fault-detection capability defined by the probability-based model is between 0 and 1. From the

[3]Using the $diff$ command, we can know the position that the added statement is supposed to be added in P_0.

preceding equation, the fault-detection capability of a test case is defined based on how likely this test case detects faults in each statement averagely. The higher probability a test case has on detecting faults in each statement, the larger its fault-detection capability is.

III. Experimental Study

To evaluate the proposed approach, we perform an experimental study to answer the following research questions.

1) **RQ1:** Is the proposed approach more effective than the total and the additional coverage-based approaches, when prioritizing test cases at different levels (i.e., test-method level or test-class level)?
2) **RQ2:** Which model (i.e., statistics-based model or probability-based model) makes the proposed approach more effective, when prioritizing test cases at different levels (i.e., test-method level or test-class level)?

A. Subjects, Test Cases, and Faults

In the experiment, we used 14 versions of 3 Java projects, which have been widely used in the literature of test-case prioritization [26], [3]. In particular, we used 5 versions of Jgrapht[4] (JGT), 2 versions of Xmlsecurity[5] (XS), and 7 versions of Jodatime[6] (JT).

In this experimental study, we collected more than one versions for each project. For each project, we used its earliest version collected as an early version, and used its other versions as latter versions in software evolution. That is, each latter version is taken as a subject in this experimental study and test-case prioritization in software evolution aims to schedule the execution order of test cases on such subjects. Note that considering the cost of mutation testing, the proposed approach is evaluated based on an early version and its several latter versions rather than two successive versions, which will be discussed in Section IV. To simulate test-case prioritization in software evolution, we used only the test cases of the early version as the input of test-case prioritization. However, some test cases of the early version become obsolete test cases and thus cannot be used to test the latter version. Therefore, we manually removed these obsolete test cases [30].

It is hard for us to collect real faults in software evolution because programmers tend to deliver projects with few or no faults. Therefore, we manually constructed faults in software evolution by using mutation faults. As faults in software evolution usually come from the modification and mutation faults may substitute real faults in software testing [24], [31], [32], we generated mutation faults on the difference between the early version and the latter version. In particular, for each latter version, we first generated all mutants using a mutation tool Javalanche [29], and then removed the mutants whose mutation operators occur on the statements that are not in the modification between the early version and the latter version. Following this process, for each latter version, we generated a number of mutants, which can be viewed as faults induced in software evolution. Then as previous work [3], we constructed a number of multiple-fault programs, each of which is a group of five randomly selected mutants which have never been selected before. Each mutant group is viewed as a program with five faults.

[4]http://www.jgrapht.org
[5]http://xml.apache.org/security
[6]http://joda-time.sourceforge.net

TABLE I
Subjects

Subject	Source Code Under Test			Test Case		Group
	LOC	Class	Method	Class	Method	
JG 5.0	2,393	72	321	18	48	–
JG 5.1	3,248	88	393	18	45	8
JG 5.2	3,609	89	411	17	39	8
JG 5.3	3,670	90	413	17	39	8
JG 6.0	4,098	96	474	16	27	7
XS 1.2.0	16,138	154	1,077	27	97	–
XS 1.2.1	16,218	154	1,074	27	93	5
JT 1.0	20,241	177	2,561	80	2,117	–
JT 1.2	22,149	182	2,722	60	1,401	19
JT 1.3	24,001	188	3,079	60	1,401	36
JT 1.4	25,292	196	3,242	60	1,396	106
JT 1.5	25,795	198	3,279	43	1,082	85
JT 1.5.2	25,807	198	3,276	43	1,081	13
JT 1.6	25,879	198	3,288	43	1,081	17

Note that the mutants used to implement the proposed approach are different from the mutants used in the evaluation. In the former process we used mutants generated based on the early version, whereas in the latter process we used mutants generated based on the latter version due to the following reasons. Test-case prioritization in software evolution is proposed to improve the testing efficiency of the latter version by scheduling the execution order of test cases. Therefore, the input of test-case prioritization intuitively cannot contain the execution information on the latter version, even if the execution information of the mutants on the latter version. That is, to guide test-case prioritization, we cannot use the mutants generated for the latter version, but the mutants generated for the early version. On the other side, test-case prioritization proposed in this paper targets at software evolution, and thus the prioritized test cases should be evaluated on the latter version. Therefore, the mutants used in the evaluation should be generated based on the latter version.

Table I presents the basic statistics of these subjects, whose last three columns present the statistics on the test code actually used in the experimental study and the number of mutant groups. As the mutant groups are constructed to simulate faults for latter versions, the number of mutant groups for each early version is marked as "–" in the table.

B. Independent Variables

The independent variables include the compared approaches and the granularities of test cases.

• **Compared Approaches:** Besides the proposed approach based on the statistics-based model and the proposed approach based on the probability-based model, we also implemented the total and additional approaches, as they are widely used as the control technique in test-case prioritization [3]. Moreover, as discovered so far [3], none of these existing test-case prioritization approaches outperform the additional strategy. In particular, the total coverage-based test-case prioritization approach schedules the execution order of test cases based on the descendent order of their coverage information (e.g., statement coverage), whereas the additional coverage-based test-case prioritization approach schedules the execution order of test cases based on the descendent order of their incremental coverage information (e.g., statement coverage) [17]. Furthermore, to learn the upper bound of test-case prioritization, we implement **the optimal test-case prioritization approach** [23], which prioritizes test cases based on the descendent order of faults they actually detect. Note that this optimal approach is not a practical approach, because it is impossible to know which faults will be detected by each test case before running these test cases. That is, this optimal approach also serves as the control approach in the evaluation.

• **Granularities of Test Cases:** The subjects used in the experimental study are all written in Java and are tested using the JUnit testing framework. In the JUnit testing framework, there are two levels of test cases: test-method level and test-class level. To investigate the impact of test-case granularities, similar to previous work [3], in this experimental study we evaluate the effectiveness of the proposed test-case prioritization using test cases at the test-method level and test cases at the test-class level.

C. Dependent Variables

The dependent variable concerns with the effectiveness measurement. In the experiment, we use Average Percentage Faults Detected (abbreviated as APFD) [23] as a measurement, which is widely used in the prior work of test-case prioritization [5], [7]. For any prioritized test suite T', the larger its APFD value is, the more effective the corresponding test-case prioritization approach is.

For any test suite T, its prioritized test suite denoted as T', we calculate the APFD value of T' as follows.

$$APFD = (\frac{\sum_{i=1}^{n-1} F_i}{n * m} + \frac{1}{2n}) * 100\% \tag{4}$$

In the equation, n is the number of test cases in T', m is the number of faults detected by the test cases in T, F_i is the number of faults detected by at least one test case among the first i test case in T'. For any prioritized test suite T', the larger its APFD value is, the more effective the corresponding test-case prioritization approach is.

D. Process

For each project, we took its earliest version as the early version and used the other versions as the latter versions. In software evolution, some test cases designed for an early version cannot be used to test its latter versions because compiling errors occur while running these test cases on the latter versions. These obsolete test cases [30] cannot be used to test the latter version directly, and thus the authors of this paper manually removed these test cases before evaluating the proposed approach.

For each latter version we applied the five test-case prioritization approaches to the set of test cases based on the following process.

As the input of the total and the additional approaches is the statement coverage of test cases, we first collected the statement coverage of test cases on the early version by instrumention, which is implemented with the ASM bytecode manipulation framework[7]. Based on the statement coverage, we applied the total and the additional approaches to the test cases at the test-method level as well as the test cases at the test-class level, and recorded the prioritized results (i.e., prioritized test cases).

Furthermore, to each set of test cases we applied the proposed mutation-based test-case prioritization approach. In implementing the proposed approach, we used the “diff” command provided by Linux Operating System to identify the different statements between each pair of early version and latter version. Moreover, we use a mutation testing tool Javalanche [29] to generate mutation faults for the early versions, some of which are used to guide test-case prioritization.

To learn whether the compared test-case prioritization approaches are effective, we constructed a number of faulty versions based on each latter version to simulate real faults in software evolution based on the following process. First, we generated all the mutants for each latter version using Javalanche. As faults in software evolution mostly lie in the difference between the two versions, we removed the mutants whose mutation operators occur on not the difference between the versions. That is, only the mutants whose mutation operators occur on the difference are taken as possible faults induced in software evolution. To construct multiple-fault programs, we randomly selected five mutants that are not selected before and constructed a mutant group consisting of five mutants. Each mutant group is viewed as a multiple-fault latter version. Following this random selection process, we constructed a number of multiple-fault latter version.

Based on each multiple-fault program, we calculated the APFD value for a prioritized result. Furthermore, we applied the optimal test-case prioritization approach to each set of test cases based on the faults within each multiple-fault program.

E. Threats to Validity

The main threat to internal validity lies in the implementation of the compared prioritization approaches. To reduce this threat, the authors of this paper reviewed all the code to guarantee its correctness.

The main threat to external validity lies in the subjects, faults, and test cases used in the experimental study. To reduce the threat resulting from the subjects, we used several large Java projects, which have been widely used in prior work [3]. However, these subjects may not be representative for other programs, especially programs in other languages. To reduce the threat resulting from the faults, we used mutation faults to simulate real faults because existing work shows that the

[7]http://asm.ow2.org

mutation faults may be valid substitutes for real faults [24], [25]. Furthermore, to simulate faults in software evolution, we used mutation faults whose mutation operators occur on only the difference between versions. To reduce the threat resulting from the test cases, we used the test cases suited by each project. However, manually removing obsolete test cases may induce external threat.

The main threat to construct validity lies in the measurement used in the experimental study. To reduce this threat, we used APFD measurement, which is widely used in test-case prioritization [5], [7]. However, considering the impact of fault severity and execution time of test cases, we will use other measurement like $APFD_c$ [33] in the future.

F. Results and Analysis

In this section, we first present the distribution of APFD results in Section III-F1 and then conduct statistics analysis on the results in Section III-F2.

1) APFD Results: Figure 1 and Figure 2 present the boxplots on the APFD results of the five test-case prioritization approaches using test cases at the test-method level and test cases at the test-class level, where the horizonal axis represents the five test-case prioritization approaches and the vertical axis represents the APFD results. For simplicity, we use "OP" to refer to the optimal approach, "C-T" refer to the total approach, "C-A" refer to the additional approach, "S-M" refer to the proposed approach using the statistics-based model, and "P-M" refer to the proposed approach using the probability-based model. Furthermore, the caption under each sub-figure in the form of "subject x1-x2 CLASS or METHOD" refers to the fact that x1 and x2 are used as the early and latter versions in the experiment. For example, the first sub-figure has a caption "Jgrapht 5.0-5.1 CLASS", which means that in the corresponding experiment the early version is Jgrapht 5.0 and the latter version is Jgrapht 5.1, and the test cases used in the experiment are at the test-class level.

From Figure 2, for the four versions of Jgrapht, the proposed approach using either the statistics-based model or the probability-based model, is always better than both the total and additional approaches when prioritizing test cases at the test-method level. Besides, the median of the proposed approach is often higher, at least not worse, than the median of the coverage-based approaches (i.e., the total and additional approaches). Moreover, the APFD distribution of the proposed approach is more centralized than the latter. Thus, for Jgrapht, the proposed approach using either the statistics-based model or the probability-based model is better than than the coverage-based approaches when prioritizing test cases at the test-method level. From Figure 2, when prioritizing test cases at the test-class level, the proposed approach with the statistics-based model outperforms the additional approach slightly in most cases. Based on these observations, we guess that the proposed approach may be more effective when prioritizing test cases in the test-method level than in the test-class level. Furthermore, for Jgrapht, the proposed approach with either the statistics model or the probability-based model achieves similar APFD results in most cases, but the former is better in the remaining cases.

For Xmlsecurity, the additional approach seems slightly better than the proposed approach, especially when prioritizing test cases at the test-method level. However, based on the median APFD results, the two approaches are similar. Considering the APFD distribution, the additional approach is more centralized than the proposed approach. We suspect the reason for this observation to be that Xmlsecurity has the smallest number of mutant groups, as shown by Table I.

For Jodatime, the proposed approach with the statistics-based model outperforms the total and additional approaches very much when prioritizing test cases at the test-method level, and the former slightly outperforms the latter when prioritizing test cases at the test-class level. In particular, in most cases, the proposed approach with the statistics-based model is much better than the additional approach when measuring their median and distribution of APFD results. In the remaining cases, the median APFD results of the proposed approach with the statistics-based model are at least the same as the additional approach. In summary, the proposed approach with the statistics-based model is better than the additional approach in most cases when prioritizing test cases at the test-class level. Moreover, when prioritizing test cases at the test-method level, the superiority of the proposed approach with the statistics-based approach over the additional approach is more obvious. Furthermore, the proposed approach with the statistics-based model is better than the proposed approach with the probability-based model in most cases, when prioritizing test-cases at the test-method level. As Jodatime is the largest projects used in the experimental study, the preceding results may show that the proposed approach is competitive.

In Figure 1 and Figure 2, we used the first box in each sub-figure to represent the APFD results of the optimal test-case prioritization approach, which serves as the upper bound of test-case prioritization. The APFD results of the other compared approaches are usually far from those of the optimal approach, although in some cases (e.g., prioritizing test cases at the test-method level for Jodatime 1.5.2) the proposed approach with the statistics-based model is very close to the latter.

To learn the impact of code change in test-case prioritization, we draw Figure 3 to represent the percentage of code change in software evolution. In particular, the horizontal axis represents the change in the form of "x1-x2" which represents x1 is changed to x2, and the vertical axis represents the percentage of code change based on the origin version. For example, the column denoted as Jodatime 1.0-1.2 represents the percentage of code change from Jodatime 1.0 to Jodatime 1.2. Through this figure, in software evolution, the percentage of code change tends to increase, which is as expected. Furthermore, from Figure 2 to Figure 3, the total and additional approaches based statement coverage tend to become less effective when more changes occur in software evolution. This observation is as expected because the total and additional approaches schedule the execution order of test cases based

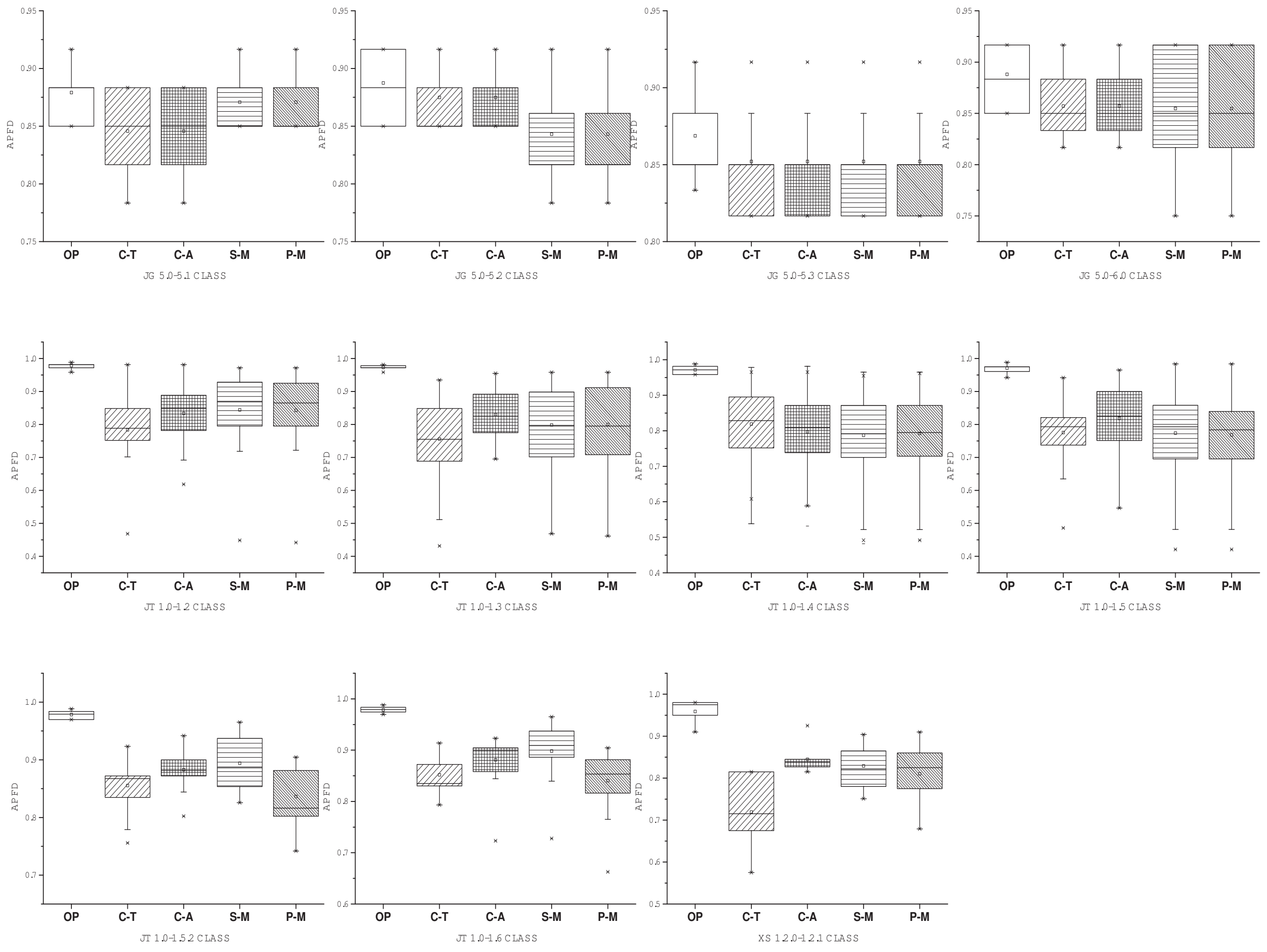

Fig. 1. APFD Results of Prioritized Test Cases at the Test-Class Level

on their coverage of the early version and non-trivial change from the early version to the latter version may definitely hamper the effectiveness of these prioritization approaches. On the contrary, our proposed approach, especially our proposed approach with the statistics-based model, is steady, even if non-trivial changes occur in software evolution. For example, in the APFD result of Jodatime on test-method level, when subject evolves from version 1.2 to version 1.6, the APFD results of the proposed approach are more stable than those of the coverage-based approaches. We suspect the reason for this observation to be that our proposed approach relies mostly on mutant killing information, which is more related to the change between the early version and the latter version. As our approach outperforms the total and additional approach in terms of stableness, our approach seems to be promising.

2) Statistics Analysis: In order to investigate whether there is significant difference between the compared approaches, we performed a sign test [34] on the APFD results of these approaches using the statistics software SPSS 15.0[8]. The results of the sign test are shown by Table II.

From this table, when prioritizing test-cases at the test-class level, the proposed approach with the statistics-based model is not significantly different from the proposed approach with the probability-based model. However, when prioritizing test cases at the test-method level, the proposed approach with the statistics-based model significantly outperforms the proposed approach with the probability-based model. Furthermore, considering the observation on Figure 1 and Figure 2, the proposed approach with the statistics-based model is significantly better than the proposed approach with the probability-based model when prioritizing test cases at the test-method level, whereas when prioritizing test cases at the test-class level, the former outperforms the latter without significant difference.

When prioritizing test cases at the test-class level, the proposed approach with the statistics-based model is not significantly different from the total approach. When prioritizing

[8]http://www.spss.com

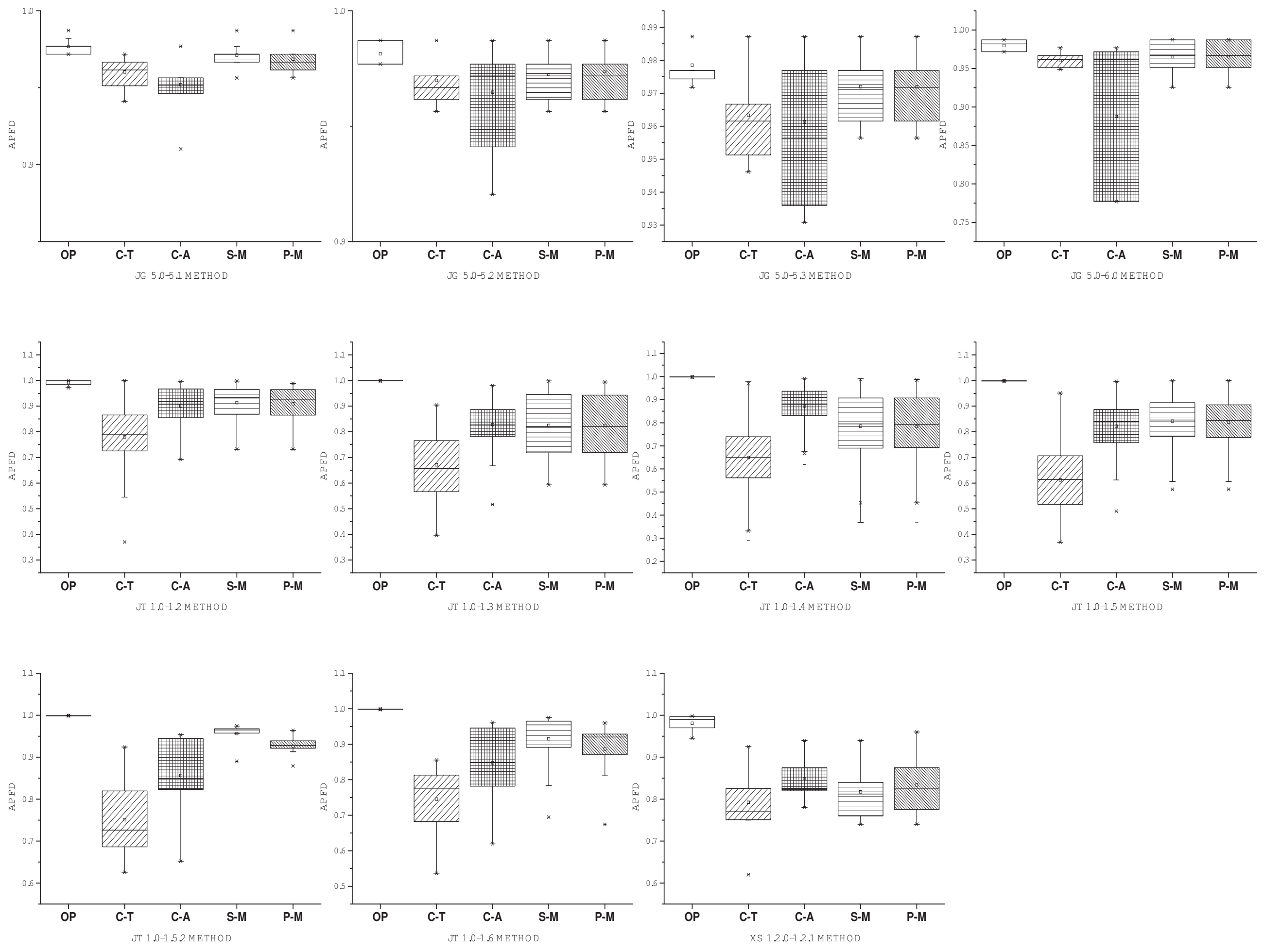

Fig. 2. APFD Results of Prioritized Test Cases at the Test-Method Level

test cases at the test-method level, the proposed approach with the statistics-based model is significantly better than the total approach. Furthermore, considering the observation on Figure 1 and Figure 2, the proposed approach with the statistics-based model is significantly better than the total approach when prioritizing test-cases at the test-method level, and is mostly better than the total approach without significant difference when prioritizing test cases at the test-class level.

When prioritizing test cases at the test-class level, the proposed approach with the statistics-based model is significantly worse than the additional approach. When prioritizing test-cases at the test-method level, the proposed approach with the statistics-based model is not significantly different from the additional approach. Considering the observation on Figure 1 and Figure 2, the proposed approach with the statistics-based model is significantly worse than the additional approach when prioritizing test cases at the test-class level, but is mostly better than the additional approach without significant difference when prioritizing test cases at the test-method level.

TABLE II
SIGH TEST ON APFD RESULTS (α=0.05)

Comparison	Granularity	n_+	n_-	p-value	Result
cov_total v.s.	test-class	143	156	0.488	Not
mutation S-M	test-method	43	264	0.000	**Reject**
cov_total v.s.	test-class	153	147	0.733	Not
mutation P-M	test-method	44	265	0.000	**Reject**
cov_add v.s.	test-class	171	128	0.015	**Reject**
mutation S-M	test-method	140	162	0.781	Not
cov_add v.s.	test-class	183	120	0.000	**Reject**
mutation P-M	test-method	146	156	0.728	Not
mutation S-M v.s.	test-class	96	119	0.134	Not
mutation P-M	test-method	179	66	0.000	**Reject**

Although the proposed approach is less effective when prioritizing test cases at the test-class level, the results may not be representative because the total number of test cases at the test-class level is small (usually smaller than 50), as shown by Table I.

3) Conclusions: According to the preceding analysis, we get the following conclusions for the two research questions.

- Our approach with the statistics-based model is signifi-

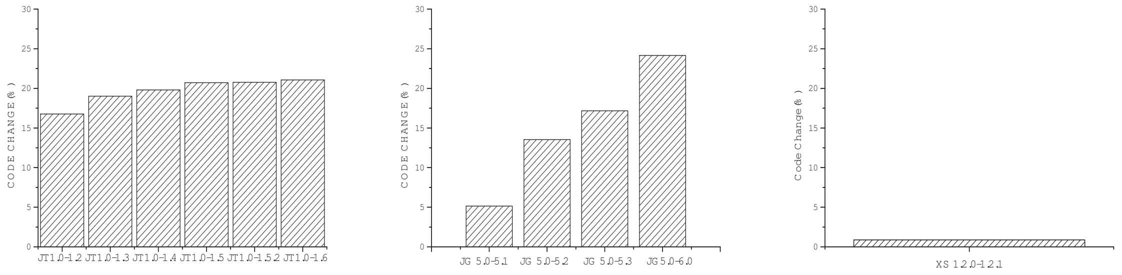

Fig. 3. Percentage of Code Change

cantly better than our approach with the probability-based model when prioritizing test cases at the test-method level, but the former outperforms the latter without significant difference when prioritizing test cases at the test-class level.

- Our approach with the statistics-based model is mostly better than the additional approach without significant difference when prioritizing test cases at the test-method level, but the former is significantly worse than the latter when prioritizing test cases at the test-class level.
- Our approach with the statistics-based model is significantly better than the total approach when prioritizing test cases at the test-method level, but the former outperforms the latter without significant difference when prioritizing test cases at the test-class level.

IV. Discussion

First, compared with the existing coverage-based approaches, the proposed approach tends to be effective but less efficient. The cost of either the coverage-based approaches or the proposed approaches includes two parts: information collection and prioritization. Considering the cost on information collection, the former approach is more efficient than the propose approach due to the following reason. To collect the coverage information required for the former approaches, developers have to instrument the program under test and run the instrumented program. The proposed approach does not require this coverage information, but the information whether each mutant is killed by each test case, which tends to be much more costly than the coverage-based approaches. Considering the cost on prioritization, the former approach is close to the proposed approach. For example, when prioritizing the test cases of Jodatime 1.0 for Jodatime-1.6, the prioritization time of the total approach is 40,155ms, whereas the prioritization time of the proposed approach is 32,500ms. Generally speaking, the proposed approach is less efficient. However, as mutation faults may simulate real faults in software evolution and thus the proposed mutation-based approach tends to be promising in terms of effectiveness. Furthermore, according to the experimental results, the proposed approach is still effective when the latter version is much different from the early version. That is, considering the efficiency issue, we may use the mutant information for an early version for a series of latter version when applying the proposed mutation-based approach. Therefore, we plan to investigate the effectiveness of the proposed approach when the two versions have much difference.

Second, the proposed approach is represented in the granularity of statements, although it can be represented in other granularities (e.g., methods). In particular, the proposed approach identifies the change between versions based on the statement granularity and then generates mutants towards these changing statements. Obviously, the proposed approach can also be represented in other granularities like methods and classes. It is less costly to identify the change in the coarse granularity (e.g., methods or files) than the fine granularity (e.g., statements), but mutant generation (which is the second step of our approach) based on the coarse-granularity change may have some shortcomings. First, mutant generation based on the coarse-granularity (e.g., methods) may cause extra efforts. Existing mutation tools produce mutants by performing some mutation operators on some statements or expressions. To generate mutants covering some method, programmers have to first identify which statements belong to this method and then generate mutants for this method (i.e., actually for the statement in this method). Second, mutant generation based on the coarse-granularity may produce imprecise data. Supposed that statement s is changed from P_0 to P_1, programmers identify the change by the method m because m contains s. In mutant generation based on changing methods rather than changing statements, programmers tend to use the mutants whose mutation operation occur in m and thus some mutant whose mutation operator occurs on the other statements of m rather than s may be selected. That is, in mutant-generation process, some mutants that are not related to the change between P_0 and P_1 may be selected and thus may decrease the effectiveness of test-case prioritization. Therefore, although many other granularities exist when representing the proposed approach, the granularity of statements is still a good choice.

Third, the proposed approach maps the change between two versions on the source code of the early version due to the following concern. As our approach is proposed in the testing scenario of software evolution, the source code of the early version and the latter versions is available. That is, we can mark the changes between versions either on the early version or the latter version. However, the mutants used to guide test-case prioritization have to be generated and ran on the early version for the sake of reducing the testing cost on the latter version. To find the mutants related to the change, it is natural to map the change between versions on the early version rather than the latter version. However, we may investigate the effectiveness of mutation-based test-case prioritization by mapping the change on the latter version in the future.

Fourth, the statement added to P_0 is mapped to the existing statement in P_0 that is closet. That is, currently the proposed approach maps the changing statements just based on their locations in the program. This strategy is not perfect and can be improved. In our future, we will identify more statements that the changing statements are control or data dependent on and use the set of these statements as $\triangle P$ to improve the proposed approach.

V. Related Work

A. Test-Case Prioritization

The existing approaches on test-case prioritization can be mainly classified two groups: general test-case prioritization and version-specific test-case prioritization [17].

General test-case prioritization does not consider the difference between versions and prioritizes test cases based on only the information of an early version. Most of the existing test-case prioritization approaches [17], [7], [22], [23], [16] belong to this category, which usually schedule the order of test cases based on some structural coverage (e.g., statement coverage, branch coverage, modified condition/decision coverage [22], [16]) of test cases on the early version. For example, Jiang [35] proposed an adaptive random approach, which selects test cases based on the distance between selected test cases and remaining unselected test cases. Zhang et al. [21] presented a unified test-case prioritization approach, which combines the benefits of both the total and the additional approach by a unified model. Rothermel et al. [17] proposed to prioritize test cases based on the number of mutants these test cases kill. Similar to our work, this work schedules the execution order of test cases based on mutants, but their mutants are generated based on the early version, some of which are definitely not related to the faults in software evolution. Furthermore, some test-case prioritization approaches [36], [11], [14], [37] are proposed by considering the constraints in software testing, e.g., time limit. These general approaches prioritize test cases only based on their information of an early version, and thus may not be effective when there is unignorable difference between the early version and the latter version.

Version-specific test-case prioritization considers the difference between versions and thus the prioritized results are supposed to be effective for some latter version, not for all the versions. For example, Wong et al. [19] proposed a modification-based approach, which analyzes the source code change and schedules test cases based on the "increasing cost per additional coverage" [19]. Srivastava and Thiagarajan [13] proposed to prioritize test cases based on their coverage on the modified binary code. Furthermore, Korel et al. [12], [20] proposed a series of model-based approaches, which schedule the order of test cases based on the modification on the system model and its execution information. Similar to these approaches, our work is also a version-specific test-case prioritization approach. However, the existing approaches rely on the structural coverage on the difference between versions, whereas our approach relies on the mutation faults simulating real faults in software evolution.

B. Mutation Testing

Mutation testing [38], [39], was initially proposed to measure the adequacy of test cases by mutants, and thus many previous researchers focus on investigating the generation of mutants [40], [41]. However, as it is costly to run to execute test suites on all the generated mutants [42], some researchers began to investigate how to reduce the cost of mutation testing mainly through two ways, mutant selection [42], [43], [44] and time reduction [45], [46], [47]. Furthermore, as mutation faults can be substitute for real faults, mutation testing is also be applied to solve other problems in software testing, e.g., bug fixing. Recently, Shi et al. [48] proposed to use the number of mutants a test case kills as a complement criterion in test case selection. Their work is similar to our work in this paper, but our work targets at prioritizing test cases in software evolution.

VI. Conclusions

In this paper we propose a novel mutation-based test-case prioritization approach for software evolution, which first generates mutation faults whose mutation operators occur on only the changed statements, and schedules the execution order of test cases based on the descendent order of their fault-detection capability, which is calculated by statistics-based model and probability-based model. From the experimental study, the proposed approach with the statistics-based model outperforms the proposed approach with the probability-based model and the total approach. Compared with the additional approach, the proposed approach is competitive.

In the future, we will improve the proposed approach and fully evaluate the proposed approach as follows. First, we will improve the proposed approach by considering program dependency in mutant selection and defining other models to measure the fault-detection capability of test cases. Second, we will evaluate the effectiveness of the proposed approach by using the closet two versions as the early version and the latter version. Third, we will evaluate the effectiveness of test-case prioritization by considering other factors like time cost and fault severity.

Acknowledgments

This work is supported by the High-Tech Research and Development Program of China under Grant No.2013AA01A605, the National Natural Science Foundation of China under Grant Nos. 61421091, 61432001, and 61272157.

REFERENCES

[1] M. M. Lehman, "Programs, life cycles, and laws of software evolution," *Proceedings of IEEE*, pp. 1060–1076, 1980.

[2] Z. Zheng, S. Ma, W. Li, W. Wei, X. Jiang, Z. Zhang, and B. Guo, "Dynamical characteristics of software trustworthiness and their evolutionary complexity," *SCIENCE CHINA Information Sciences*, vol. 52, no. 8, pp. 1328–1334, 2009.

[3] H. Mei, D. Hao, L. Zhang, L. Zhang, J. Zhou, and G. Rothermel, "A static approach to prioritizing JUnit test cases," *IEEE Transactions on Software Engineering*, vol. 38, no. 6, pp. 1258–1275, 2012.

[4] H. Do, G. Rothermel, and A. Kinneer, "Empirical studies of test case prioritization in a JUnit testing environment," in *Proceedings of the International Symposium on Software Reliability Engineering*, 2004, pp. 113–124.

[5] S. Elbaum, A. Malishevsky, and G. Rothermel, "Prioritizing test cases for regression testing," in *Proceedings of the International Symposium on Software Testing and Analysis*, 2000, pp. 102–112.

[6] ——, "Incorporating varying test costs and fault severities into test case prioritization," in *Proceedings of the International Conference on Software Engineering*, 2001, pp. 329–338.

[7] ——, "Test case prioritization: a family of empirical studies," *IEEE Transactions on Software Engineering*, vol. 28, no. 2, pp. 159–182, 2002.

[8] S.-S. Hou, L. Zhang, T. Xie, and J. Sun, "Quota-constrained test-case prioritization for regression testing of service-centric systems," in *Proceedings of the International Conference on Software Maintenance*, 2008, pp. 257–266.

[9] B. Jiang, Z. Zhang, W. K. Chan, and T. H. Tse, "Adaptive random test case prioritization," in *Proceedings of Automated Software Engineering*, 2009, pp. 257–266.

[10] J. A. Jones and M. J. Harrold, "Test-suite reduction and prioritization for modified condition/decision coverage," in *Proceedings of the International Conference on Software Maintenance*, 2001, pp. 92–101.

[11] J. M. Kim and A. Porter, "A history-based test prioritization technique for regression testing in resource constrained environments," in *Proceedings of the International Conference on Software Engineering*, 2002, pp. 119–129.

[12] B. Korel, L. Tahat, and M. Harman, "Test prioritization using system models," in *Proceedings of the International Conference on Software Maintenance*, 2005, pp. 559–568.

[13] A. Srivastava and J. Thiagarajan, "Effectively prioritizing tests in development environment," in *Proceedings of the International Symposium on Software Testing and Analysis*, 2002, pp. 97–106.

[14] K. R. Walcott, M. L. Soffa, G. M. Kapfhammer, and R. S. Roos, "Time aware test suite prioritization," in *Proceedings of the International Symposium on Software Testing and Analysis*, 2006, pp. 1–11.

[15] L. Zhang, J. Zhou, D. Hao, L. Zhang, and H. Mei, "Prioritizing JUnit test cases in absence of coverage information," in *Proceedings of the International Conference on Software Maintenance*, 2009, pp. 19–28.

[16] C. Fang, Z. Chen, and B. Xu, "Comparing logic coverage criteria on test case prioritization," *Science China: Information Science*, vol. 55, no. 12, pp. 2826–2840, 2012.

[17] G. Rothermel, R. J. Untch, C. Chu, and M. J. Harrold, "Prioritizing test cases for regression testing," *IEEE Transactions on Software Engineering*, vol. 27, no. 10, pp. 929–948, 2001.

[18] D. Hao, X. Zhao, and L. Zhang, "Adaptive test-case prioritization guided by output inspection," in *Proceedings of the 37th Annual IEEE Computer Software and Applications Conference*, 2013, pp. 169–179.

[19] W. Wong, J. Horgan, S. London, and H. Agrawal, "A study of effective regression testing in practice," in *Proceedings of the 8th International Symposium on Software Reliability Engineering*, 1997, pp. 230–238.

[20] B. Korel, G. Koutsogiannakis, and L. Tahat, "Application of system models in regression test suite prioritization," in *Proceedings of the International Conference on Software Maintenance*, 2008, pp. 247–256.

[21] L. Zhang, D. Hao, L. Zhang, G. Rothermel, and H. Mei, "Bridging the gap between the total and additional test-case prioritization strategies," in *Proceedings of the International Conference on Software Engineering*, 2013, pp. 192–201.

[22] J. A. Jones and M. J. Harrold, "Test-suite reduction and prioritization for modified condition/decision coverage," *IEEE Transactions on Software Engineering*, vol. 29, no. 3, pp. 195–209, 2003.

[23] G. Rothermel, R. H. Untch, C. Chu, and M. J. Harrold, "Test case prioritization: an empirical study," in *Proceedings of the International Conference on Software Maintenance*, 1999, pp. 179–188.

[24] J. H. Andrews, L. C. Briand, and Y. Labiche, "Is mutation an appropriate tool for testing experiments?" in *Proceedings of the International Conference on Software Engineering*, 2005, pp. 402–411.

[25] R. Just, D. Jalali, L. Inozemtseva, M. D. Ernst, R. Holmes, and G. Fraser, "Are mutants a valid substitute for real faults in software testing?" in *Proceedings of the Symposium on the Foundations of Software Engineering*, 2014.

[26] D. Hao, L. Zhang, L. Zhang, G. Rothermel, and H. Mei, "A unified test-case prioritization approach," *ACM Transactions on Software Engineering and Methodology*, vol. 24, no. 2, p. 10, 2014.

[27] A. J. Offutt, G. Rothermel, and C. Zpf, "An experimental evaluation of selective mutation," in *Proceedings of the International Conference on Software Engineering*, 1993, pp. 100–107.

[28] J. Zhang, M. Zhu, D. Hao, and L. Zhang, "An empirical study on the scalability of selective mutation testing," in *Proceedings of the 25th International Symposium on Software Reliability Engineering*, 2014, pp. 277–287.

[29] D. Schuler and A. Zeller, "Javalanche: efficient mutation testing for Java," in *Proceedings of the ACM Symposium on Foundations of Software Engineering*, 2009, pp. 297–298.

[30] D. Hao, T. Lan, H. Zhang, C. Guo, and L. Zhang, "Is this a bug or an obsolete test?" in *Proceedings of the 27th European Conference on Object-Oriented Programming*, 2013, pp. 602–628.

[31] J. H. Andrews, L. C. Briand, Y. Labiche, and A. Siami Namin, "Using mutation analysis for assessing and comparing testing coverage criteria," *IEEE Transactions on Software Engineering*, vol. 32, no. 8, pp. 608–624, 2006.

[32] H. Do and G. Rothermel, "On the use of mutation faults in empirical assessments of test case prioritization techniques," *IEEE Transactions on Software Engineering*, vol. 32, no. 9, pp. 733–752, 2006.

[33] A. Malishevsky, J. R. Ruthruff, G. Rothermel, and S. Elbaum, "Cost-cognizant test case prioritization," Department of Computer Science and Engineering, University of Nebraska, Tech. Rep., 2006.

[34] J. Newmark, *Statistics and Probability in Modern Life*. Philadelphia: Saunders College Pub, 1992.

[35] B. Jiang, Z. Zhang, W. K. Chan, and T. H. Tse, "Adaptive random test-case prioritization," in *Proceedings of the 24th IEEE/ACM International Conference on Automated Software Engineering*, 2009, pp. 257–266.

[36] L. Zhang, S. Hou, C. Guo, T. Xie, and H. Mei, "Time-aware test-case prioritization using integer linear programming," in *Proceedings of the International Symposium on Software Testing and Analysis*, 2009, pp. 213–224.

[37] H. Park, H. Ryu, and J. Baik, "Historical value-based approach for cost-cognizant test case prioritization to improve the effectiveness of regression testing," in *Proceedings of the International Conference on Secure Software Integration and Reliability Improvement*, 2008, pp. 39–46.

[38] R. A. DeMillo, R. J. Lipton, and F. G. Sayward, "Hints on test data selection: Help for the practicing programmer," *Computer*, vol. 11, no. 4, pp. 34–41, 1978.

[39] R. G. Hamlet, "Testing programs with the aid of a compiler," *IEEE TSE*, no. 4, pp. 279–290, 1977.

[40] A. J. Offutt, A. Lee, G. Rothermel, R. H. Untch, and C. Zapf, "An experimental determination of sufficient mutant operators," *ACM Transactions on Software Engineering and Methodology*, vol. 5, no. 2, pp. 99–118, 1996.

[41] E. F. Barbosa, J. C. Maldonado, and A. M. R. Vincenzi, "Toward the determination of sufficient mutant operators for C," *Software Testing, Verification, and Reliability*, vol. 11, no. 2, pp. 113–136, 2001.

[42] L. Zhang, S.-S. Hou, J.-J. Hu, T. Xie, and H. Mei, "Is operator-based mutant selection superior to random mutant selection?" in *Proceedings of the International Conference on Software Engineering*, 2010, pp. 435–444.

[43] L. Zhang, M. Gligoric, D. Marinov, and S. Khurshid, "Operator-based and random mutant selection: Better together," in *Proceedings of the 30th International Conference on Automated Software Engineering*, 2013, pp. 92–102.

[44] K. Adamopoulos, M. Harman, and R. M. Hierons, "How to overcome the equivalent mutant problem and achieve tailored selective mutation using co-evolution," in *Proceedings of the Genetic and Evolutionary Computation Conference*, 2004, pp. 1338–1349.

[45] R. H. Untch, A. J. Offutt, and M. J. Harrold, "Mutation analysis using mutant schemata," in *Proceedings of the International Symposium on Software Testing and Analysis*, vol. 18, no. 3, 1993, pp. 139–148.

[46] L. Zhang, D. Marinov, and S. Khurshid, "Faster mutation testing inspired by test prioritization and reduction," in *Proceedings of the International Symposium on Software Testing and Analysis*, 2013, pp. 235–245.

[47] L. Zhang, D. Marinov, L. Zhang, and S. Khurshid, "Regression mutation testing," in *Proceedings of the International Symposium on Software Testing and Analysis*, 2012, pp. 331–341.

[48] M. G. A. Z. D. M. August Shi, Alex Gyori, "Balancing trade-offs in test-suite reduction," in *Proceedings of the 22nd ACM SIGSOFT Intenational Symposium on Foundations of Software Engineering*, 2014, pp. 246–256.