To Be Optimal Or Not in Test-Case Prioritization
Dan Hao, Lu Zhang, Lei Zang, Yanbo Wang, Xingxia Wu, Tao Xie

Abstract—Software testing aims to assure the quality of software under test. To improve the efficiency of software testing, especially regression testing, test-case prioritization is proposed to schedule the execution order of test cases in software testing. Among various test-case prioritization techniques, the simple additional coverage-based technique, which is a greedy strategy, achieves surprisingly competitive empirical results. To investigate how much difference there is between the order produced by the additional technique and the optimal order in terms of coverage, we conduct a study on various empirical properties of optimal coverage-based test-case prioritization. To enable us to achieve the optimal order in acceptable time for our object programs, we formulate optimal coverage-based test-case prioritization as an integer linear programming (ILP) problem. Then we conduct an empirical study for comparing the optimal technique with the simple additional coverage-based technique. From this empirical study, the optimal technique can only slightly outperform the additional coverage-based technique with no statistically significant difference in terms of coverage, and the latter significantly outperforms the former in terms of either fault detection or execution time. As the optimal technique schedules the execution order of test cases based on their structural coverage rather than detected faults, we further implement the ideal optimal test-case prioritization technique, which schedules the execution order of test cases based on their detected faults. Taking this ideal technique as the upper bound of test-case prioritization, we conduct another empirical study for comparing the optimal technique and the simple additional technique with this ideal technique. From this empirical study, both the optimal technique and the additional technique significantly outperform the ideal technique in terms of coverage, but the latter significantly outperforms the former two techniques in terms of fault detection. Our findings indicate that researchers may need take cautions in pursuing the optimal techniques in test-case prioritization with intermediate goals.

Index Terms—Test-Case Prioritization, Integer Linear Programming, Greedy Algorithm, Empirical Study.

1 INTRODUCTION
Regression testing is an expensive task in software maintenance. For example, the industrial collaborators of Elbaum et al. [1], [2] reported that it costs seven weeks to execute the entire test suite of one of their products. Test-case prioritization [3], [4], [5], [6], [7], [8], whose aim is to maximize a certain goal of regression testing via re-ordering test cases, is an intensively investigated approach for reducing the cost of regression testing.

As the main goal of regression testing is to assure the quality of the software under regression testing, most techniques for test-case prioritization aim at maximizing the fault-detection capability of the prioritized set of test cases. As whether a test case can detect a fault is unknown before running the software under test, the fault-detection capability can hardly be used to guide scheduling the execution order of the test cases directly. Since a test case cannot detect a fault if the test case does not execute (or cover) the corresponding faulty structural unit, most techniques for test-case prioritization (e.g., [9], [10], [6], [11]) use structural coverage as the substitutive goal (i.e., intermediate goal) for test-case prioritization. However, these techniques can be only sub-optimal if measured based on the intermediate goal. Moreover, the simple additional coverage-
based technique\textsuperscript{1}, which is proposed in an early paper on test-case prioritization by Rothermel et al. \cite{6}, remains very competitive among all the existing techniques for test-case prioritization \cite{4}, \cite{10}. In particular, with regard to the intermediate goal, the additional coverage-based technique is by far the most effective technique \cite{4}.

Considering the surprisingly good empirical results of the additional coverage-based technique, we are curious about how much difference there would be between the order of test cases achieved by the additional coverage-based technique and the order of test cases with the optimal value on the intermediate goal. Furthermore, as it may be very costly to guarantee optimality due to the NP-hardness of the test-case prioritization problem, it is also interesting to investigate other empirical properties of the optimal order to understand whether it is cost-effective to achieve the optimal order.

To learn the cost and effectiveness difference between the additional coverage-based technique and the optimal order, we conduct an empirical study on ten non-trivial object projects, systematically investigating empirical properties of optimal coverage-based test-case prioritization in comparison with the additional coverage-based test-case prioritization. To enable our study, we model optimal coverage-based test-case prioritization\textsuperscript{2} as an Integer Linear Programming (ILP) \cite{12} problem and thus are able to achieve the optimal order in terms of coverage in acceptable time using an existing ILP solver for many non-trivial programs. In particular, our empirical study evaluates the effectiveness and efficiency of the two techniques using three metrics. Considering the impact of coverage granularity, our empirical study further considers two types of coverage for both the optimal technique and the additional technique: statement coverage and method (function) coverage.

Furthermore, to learn the upper bound of test-case prioritization, we also implement the ideal optimal test-case prioritization technique, which schedules the execution order of test cases based on the number of detected faults. Although this ideal optimal technique is not practical, it may serve as a control technique. Then we conduct an empirical study on another five non-trivial object projects, systematically investigating the effectiveness of the optimal technique and the additional technique compared with the ideal technique.

According to our empirical results, the optimal technique is slightly better than the additional technique with ignorable difference for achieving optimal coverage. However, the optimal technique is significantly worse than the additional technique for most target programs in terms of fault detection. Moreover, although both the optimal technique and the additional technique significantly outperform the ideal technique in terms of coverage, the latter significantly outperforms the former two techniques in terms of fault detection. Therefore, in test-case prioritization, it is not worthwhile to pursue optimality by taking the coverage as an intermediate goal.

This article makes the following main contributions:

- To our knowledge, the first empirical study on the optimal coverage-based test-case prioritization, demonstrating that the optimal technique may be inferior to the additional technique in practice.
- A formulation of optimal coverage-based test-case prioritization as an integer linear programming problem, which enables us to obtain the execution order of test cases to achieve optimal coverage.

The remaining of this article is organized as follows. Section 2 presents some background of test-case prioritization. Section 3 presents a formulation of optimal coverage-based test-case prioritization as an ILP problem. Section 4 and Section 5 present the design, results, and analysis of our two empirical studies. Section 6 discusses some issues in this article. Section 7

\textsuperscript{1} The additional coverage-based technique is a greedy algorithm that always orders the test case covering the most structural units (e.g., statements or methods) not yet covered by previously executed test cases before any other previously unexecuted test cases.

\textsuperscript{2} Rothermel et al. \cite{6} used an optimal technique as a control technique to evaluate the effectiveness of some techniques for test-case prioritization. However, their optimal technique is a technique with the knowledge of which test case detects which fault but still using the additional strategy to order test cases.
briefly presents related work and Section 8 concludes this article.

2 BACKGROUND

In this section, we present background on test-case prioritization along with optimal test-case prioritization.

2.1 Test-Case Prioritization

Test-case prioritization [3], [4], [5], [6], [7], [8] is a typical software-engineering task in regression testing, which is formally defined [1] as follows. Given a test suite $T$ and its set of permutation on the test cases denoted $P_T$, test-case prioritization aims to find a permutation $P_S$ in $P_T$ such that for any permutation $T'$ of $P_T$, $f(P_S) \geq f(T')$, where $f$ is a function from $P_T$ to a real number that represents the fault-detection capability. That is, the ultimate goal of test-case prioritization is to maximize the early fault-detection capability of the prioritized list of test cases.

Researchers proposed APFD [6], which is the abbreviation of average percentage of faults detected, to measure the effectiveness of the prioritized list of test cases on detecting faults. Formula 1 gives the definition of APFD.

$$APFD = 1 - \frac{\sum_{j=1}^{m} TF_j}{nm} + \frac{1}{2n}$$ (1)

In Formula 1, $m$ denotes the number of faults detected by the test suite $T$, $n$ denotes the number of test cases in $T$, and $TF_j$ denotes the first test case in the $T'$ (which is a permutation of $T$) that exposes the fault $j$. As $n$ and $m$ are fixed for any given test suite and faulty program, higher APFD values imply higher fault-detection rates.

For ease of representation, we use an example program with 5 faults and a test suite with 5 test cases to explain test-case prioritization. Table 1 shows the fault-detection capability (from the second column to the sixth column) and statement coverage (from the seventh column to the last column) of these test cases. In particular, we use $*$ to represent that the corresponding fault is detected by the corresponding test case and $\bullet$ to represent that the corresponding statement is covered by the corresponding test case. If the test suite is executed in the order A-B-C-D-E, the corresponding APFD value is 0.74.

2.2 Optimal Test-Case Prioritization

As whether a test case can detect a fault is unknown before running the software under test, the fault-detection capability (measured by APFD), which can be viewed as an ultimate goal of test-case prioritization, cannot be used to guide scheduling the execution order of the test cases directly. Therefore, most techniques for test-case prioritization (e.g., [9], [10], [6], [11]) use structural coverage as the intermediate goal for test-case prioritization. Taking structural coverage as an intermediate goal, analogous to the APFD metric, Li et al. [4] proposed three coverage-based metrics, which are average percentage of branch coverage (APBC), average percentage of decision coverage (APDC), and average percentage of statement coverage (APSC), to measure the effectiveness of the prioritized set of test cases to achieve structural coverage. For the simplicity of presentation, we use a generic term (i.e., $APxC$) to represent the three coverage-based metrics and any other coverage-based metrics. Formula 2 gives the definition of $APxC$.

$$APxC = 1 - \frac{\sum_{j=1}^{u} TS_j}{nu} + \frac{1}{2n}$$ (2)

In Formula 2, $u$ denotes the number of structural units (e.g., branches, statements, or methods), $n$ denotes the number of test cases in $T$, and $TS_j$ denotes the first test case in $T'$ (which

<table>
<thead>
<tr>
<th>Table 1</th>
<th>Example Test Suite</th>
</tr>
</thead>
<tbody>
<tr>
<td>Test</td>
<td>Fault</td>
</tr>
<tr>
<td>A</td>
<td>$\ast$</td>
</tr>
<tr>
<td>B</td>
<td>$\ast$</td>
</tr>
<tr>
<td>C</td>
<td>$\ast$</td>
</tr>
<tr>
<td>D</td>
<td>$\ast$</td>
</tr>
<tr>
<td>E</td>
<td>$\ast$</td>
</tr>
</tbody>
</table>

3. Researchers also extended the APFD metric to consider other concerns (e.g., fault-severity and cost) in software testing, and proposed metrics such as $APFD_c$ [13] and Normalized APFD [14].
is a permutation of $T$) that covers the structural unit $j$. Note that given a different type of coverage, APxC denotes a different metric. For example, APxC denotes APSC for statement coverage but APMC for method (function) coverage.

In regression testing, a test case has been executed on the previous version of the software under test, and thus the structural coverage of a test case is known and can be used to directly guide scheduling the execution order of the test cases on the current version of the software under test. In particular, the simple additional coverage-based technique is proposed [6], which schedules the execution order of test cases based on the number of structural units uncovered by previously selected test cases. Given a test suite $T$, we use $T''$ to represent the set of selected test cases and then $T-T''$ represent the set of unselected test cases in the process of test-case prioritization. In this process, each time the additional coverage-based technique selects a test case $t$ from $T-T''$ so as to maximize $f(T'' \cup t)$, which represents the number of structural units covered by $T''$ and $t$. Applying the additional coverage-based technique to the test suite in Table 1, we get another execution order of this test suite D-A-E-B-C, D-A-E-C-B, D-E-A-B-C, D-E-A-C-B, D-E-C-A-B, or D-E-C-B-A.

Furthermore, taking the structure coverage as a goal, it is possible to produce an optimal order of test cases for maximizing the early structural coverage (e.g., branch coverage, decision coverage, or statement coverage). In this article, we view this process as optimal test-case prioritization. In this process, each time the additional coverage-based technique selects a test case $t$ from $T-T''$ so as to maximize $f(T'' \cup t)$, which represents the number of structural units covered by $T''$ and $t$. Applying the additional coverage-based technique to the test suite in Table 1, we get another execution order of this test suite D-A-E-B-C, D-A-E-C-B, D-E-A-B-C, D-E-A-C-B, D-E-C-A-B, or D-E-C-B-A.

For ease of presentation, we present the optimal coverage-based technique in terms of statement coverage. Analogously, the technique can be easily extended to other coverage criteria (e.g., method coverage, block coverage, and decision coverage) in test-case prioritization.

Let us consider a program containing $m$ statements (denoted as $ST = \{st_1, st_2, st_3, \ldots, st_m\}$) and a test suite containing can achieve 100% statement coverage. That is, in all the permutations of the five test cases, such an execution order achieves the maximized APSC, which is 0.77. However, such optimality does not refer to optimal APFD. Using this optimal order, executing the first two test cases A and E, only three faults are detected. Suppose that test cases B and C are executed as the first two test cases, and then all the faults can be detected. That is, using the test cases B and C as the first two test cases in the execution order may produce larger APFD values. In summary, optimal test-case prioritization mentioned in this article does not refer to the traditional optimal test-case prioritization [15], which prioritizes test cases based on the number of faults that they actually detect given that such information is known. To avoid misunderstanding, we use ideal optimal test-case prioritization to refer to traditional optimal test-case prioritization in this article. Note that ideal optimal test-case prioritization is not a practical technique because the number of faults that each test case detects is unknown before testing.

3 Optimal Coverage-based Test-case Prioritization Using ILP

As our study focuses on empirical properties of optimal coverage-based test-case prioritization, a precondition is to implement a technique for optimal coverage-based test-case prioritization to enable us to achieve optimal APxC values for at least some middle-sized programs. Inspired by our previous work [11] on test-case prioritization, we model optimal coverage-based test-case prioritization by an integer linear programming model. Using this model, we are able to achieve an optimal order of test cases for all the programs used in our study.

For ease of presentation, we present the optimal coverage-based technique in terms of statement coverage. Analogously, the technique can be easily extended to other coverage criteria (e.g., method coverage, block coverage, and decision coverage) in test-case prioritization.

Let us consider a program containing $m$ statements (denoted as $ST = \{st_1, st_2, st_3, \ldots, st_m\}$) and a test suite containing can achieve 100% statement coverage. That is, in all the permutations of the five test cases, such an execution order achieves the maximized APSC, which is 0.77. However, such optimality does not refer to optimal APFD. Using this optimal order, executing the first two test cases A and E, only three faults are detected. Suppose that test cases B and C are executed as the first two test cases, and then all the faults can be detected. That is, using the test cases B and C as the first two test cases in the execution order may produce larger APFD values. In summary, optimal test-case prioritization mentioned in this article does not refer to the traditional optimal test-case prioritization [15], which prioritizes test cases based on the number of faults that they actually detect given that such information is known. To avoid misunderstanding, we use ideal optimal test-case prioritization to refer to traditional optimal test-case prioritization in this article. Note that ideal optimal test-case prioritization is not a practical technique because the number of faults that each test case detects is unknown before testing.

3 Optimal Coverage-based Test-case Prioritization Using ILP

As our study focuses on empirical properties of optimal coverage-based test-case prioritization, a precondition is to implement a technique for optimal coverage-based test-case prioritization to enable us to achieve optimal APxC values for at least some middle-sized programs. Inspired by our previous work [11] on test-case prioritization, we model optimal coverage-based test-case prioritization by an integer linear programming model. Using this model, we are able to achieve an optimal order of test cases for all the programs used in our study.

For ease of presentation, we present the optimal coverage-based technique in terms of statement coverage. Analogously, the technique can be easily extended to other coverage criteria (e.g., method coverage, block coverage, and decision coverage) in test-case prioritization.

Let us consider a program containing $m$ statements (denoted as $ST = \{st_1, st_2, st_3, \ldots, st_m\}$) and a test suite containing can achieve 100% statement coverage. That is, in all the permutations of the five test cases, such an execution order achieves the maximized APSC, which is 0.77. However, such optimality does not refer to optimal APFD. Using this optimal order, executing the first two test cases A and E, only three faults are detected. Suppose that test cases B and C are executed as the first two test cases, and then all the faults can be detected. That is, using the test cases B and C as the first two test cases in the execution order may produce larger APFD values. In summary, optimal test-case prioritization mentioned in this article does not refer to the traditional optimal test-case prioritization [15], which prioritizes test cases based on the number of faults that they actually detect given that such information is known. To avoid misunderstanding, we use ideal optimal test-case prioritization to refer to traditional optimal test-case prioritization in this article. Note that ideal optimal test-case prioritization is not a practical technique because the number of faults that each test case detects is unknown before testing.

3 Optimal Coverage-based Test-case Prioritization Using ILP

As our study focuses on empirical properties of optimal coverage-based test-case prioritization, a precondition is to implement a technique for optimal coverage-based test-case prioritization to enable us to achieve optimal APxC values for at least some middle-sized programs. Inspired by our previous work [11] on test-case prioritization, we model optimal coverage-based test-case prioritization by an integer linear programming model. Using this model, we are able to achieve an optimal order of test cases for all the programs used in our study.

For ease of presentation, we present the optimal coverage-based technique in terms of statement coverage. Analogously, the technique can be easily extended to other coverage criteria (e.g., method coverage, block coverage, and decision coverage) in test-case prioritization.
n test cases (denoted as \( T = \{t_1, t_2, t_3, ... t_n\} \)). Let \( T' \) be a permutation of \( T \). In the following, Sections 3.1, 3.2, and 3.3 present decision variables, constraints, and the objective function in the ILP model, respectively. Section 3.4 presents the optimization to further reduce the size of the ILP model.

### 3.1 Decision Variables

The ILP model uses two groups of decision variables. Section 3.1.1 presents the first group of variables representing in what order the test cases are executed. Section 3.1.2 presents the second group of variables representing which statements are covered after the execution of each test case.

#### 3.1.1 Variables for Execution Order

To represent in what position each test case is executed, the ILP model uses \( n \times n \) Boolean decision variables (denoted as \( x_{ij} \), where \( 1 \leq i, j \leq n \)). Formally, \( x_{ij} \) is defined in Formula 3 as follows.

\[
x_{ij} = \begin{cases} 
1, & \text{if the } j\text{-th test case in } T' \text{ is } t_i \ (1 \leq i, j \leq n); \\
0, & \text{otherwise}.
\end{cases}
\]

(3)

Note that Section 3.2.1 presents a group of constraints to ensure that the set of values for \( x_{ij} \) corresponds to a permutation of \( T \).

#### 3.1.2 Variables for Statement Coverage

To represent which statements are covered after executing each test case (i.e., accumulative statement coverage), the ILP model uses \( n \times m \) Boolean decision variables (denoted as \( y_{jk} \), where \( 1 \leq j \leq n \) and \( 1 \leq k \leq m \)). Formally, \( y_{jk} \) is defined in Formula 4 as follows.

\[
y_{jk} = \begin{cases} 
1, & \text{if the first } j \text{ test cases in } T' \text{ covers } st_k \\
0, & \text{otherwise}.
\end{cases}
\]

(4)

Similarly, a group of constraints (presented in Section 3.2.2) are needed to ensure that the coverage information represented by \( y_{jk} \) (\( 1 \leq j \leq n \) and \( 1 \leq k \leq m \)) is in accordance with the execution order represented by \( x_{ij} \) (\( 1 \leq i, j \leq n \)).

### 3.2 Constraints

The ILP model uses two groups of constraints. Section 3.2.1 presents the first group of constraints to ensure that values of variables in Section 3.1.1 represent a possible execution order of test cases. Constraints in Section 3.2.1 ensure that values of variables in Section 3.1.2 are consistent with values of variables in Section 3.1.1.

#### 3.2.1 Constraints for Execution Order

There are two sets of constraints in the ILP model to ensure that values of \( x_{ij} \) (\( 1 \leq i, j \leq n \)) represent a possible execution order of test cases. First, the constraints in Formula 5 can ensure that each position in the permutation of \( T \) holds one and only one test case. Second, the constraints in Formula 6 can ensure that each test case appears in the permutation of \( T \) once and only once.

\[
\sum_{i=1}^{n} x_{ij} = 1 \quad (1 \leq j \leq n)
\]

(5)

\[
\sum_{j=1}^{n} x_{ij} = 1 \quad (1 \leq i \leq n)
\]

(6)

#### 3.2.2 Constraints for Statement Coverage

There are also a group of constraints in the ILP model to ensure that the values of \( y_{jk} \) (\( 1 \leq j \leq n \) and \( 1 \leq k \leq m \)) are in accordance with the values of \( x_{ij} \) (\( 1 \leq i, j \leq n \)). The definition of these constraints is involved with the coverage information of the test cases. In particular, for a test suite (denoted as \( T = \{t_1, t_2, ... t_n\} \)) and a set of statements (denoted as \( ST = \{st_1, st_2, ... st_m\} \)), Formula 7 is used to represent whether a test case covers a statement.

\[
c_{ik} = \begin{cases} 
1, & \text{if } t_i \text{ covers } st_k; \\
0, & \text{otherwise}.
\end{cases}
\]

(7)

Note that \( c_{ik} \) (\( 1 \leq i \leq n \) and \( 1 \leq k \leq m \)) are not variables and their values can be obtained before test-case prioritization. Based on the coverage information, the ILP model uses the constraints in Formula 8 to ensure that the values of \( y_{ik} \) (\( 1 \leq k \leq m \)) are in accordance
with the statements covered by the first test case in $T'_{i}$
\[ \sum_{i=1}^{n} c_{ik} \times x_{i1} = y_{1k} \quad (1 \leq k \leq m) \quad (8) \]

First, in Formula 8, $\sum_{i=1}^{n} c_{ik} \times x_{i1} \quad (1 \leq k \leq m)$ represents whether $s_{tk}$ is covered by the first test case in the permutation of $T$. The reason is as follows. Only one test case appears as the first test case in $T'$, and thus only one variable among $x_{i1} \quad (1 \leq i \leq n)$ is 1 according to Formula 5. As a result, for any $s_{tk}$ \quad (1 \leq k \leq m), if the first test case in $T'$ covers $s_{tk}$, $\sum_{i=1}^{n} c_{ik} \times x_{i1}$ is 1; otherwise, $\sum_{i=1}^{n} c_{ik} \times x_{i1}$ is 0. Second, based on the preceding meaning of $\sum_{i=1}^{n} c_{ik} \times x_{i1} \quad (1 \leq k \leq m)$, Formula 8 ensures that, for any $1 \leq k \leq m$, $y_{1k}$ is 1 if and only if the first test case in $T'$ covers $s_{tk}$.

Similarly, $\sum_{i=1}^{n} c_{ik} \times x_{ij} \quad (2 \leq j \leq n, 1 \leq k \leq m)$ represents whether the $j$-th test case in $T'$ covers $s_{tk}$. Thus, Formulae 9, 10, and 11 ensure that, for any $2 \leq j \leq n$ and any $1 \leq k \leq m$, $y_{jk}$ is 1 if and only if the $j$-th test case in $T'$ covers $s_{tk}$ or at least one test case before the $j$-th test case in $T'$ covers $s_{tk}$.

\[ y_{jk} \geq \sum_{i=1}^{n} c_{ik} \times x_{ij} \quad (2 \leq j \leq n, 1 \leq k \leq m) \quad (9) \]

\[ y_{jk} \geq y_{j-1,k} \quad (2 \leq j \leq n, 1 \leq k \leq m) \quad (10) \]

\[ \sum_{i=1}^{n} c_{ik} \times x_{ij} + y_{j-1,k} \geq y_{jk} \quad (2 \leq j \leq n, 1 \leq k \leq m) \quad (11) \]

That is to say, the constraints in Formulae 8, 9, 10, and 11 ensure that the values of $y_{jk} \quad (1 \leq j \leq n$ and $1 \leq k \leq m)$ are in accordance with the accumulated statement coverage for all the $n$ test cases in $T'$. Note that the variable $y_{jk}$ is a boolean variable whose value is either 0 or 1.

### 3.3 Objective Function
The objective function is to maximize the APSC metric, which is an instance of APXC (defined in Formula 2). In fact, the definition of APSC using Formula 2 is equivalent to that in Formula 12, in which $m$ denotes the number of statements, $n$ denotes the number of test cases in $T$, and $N_{i}$ denotes the number of statements covered by at least one test case among the first $i$ test cases in $T'$ (which is a permutation of $T$).

\[ \text{APSC} = \frac{\sum_{i=1}^{n-1} N_{i}}{nm} + \frac{1}{2n} \quad (12) \]

Based on this definition, maximizing the APSC value is thus equivalent to maximizing $\sum_{i=1}^{n-1} N_{i}$. As $y_{jk} \quad (1 \leq j \leq n$ and $1 \leq k \leq m$) denotes whether $s_{tk}$ is covered after executing the first $j$ test cases in $T'$, $\sum_{k=1}^{n} y_{jk} \quad (1 \leq j \leq n)$ is then the number of statements covered by at least one test case among the first $j$ test cases in $T'$. Thus, Formula 13 can serve as the objective function in the ILP model for optimal coverage-based test-case prioritization.}

\[ \text{maximize} \quad \sum_{j=1}^{n-1} \sum_{k=1}^{m} y_{jk} \quad (13) \]

### 3.4 Optimization
In the preceding ILP model, the number of decision variables is $|T| \times |T| + |ST| \times |T|$ and the number of constraints is $3 \times |ST| \times |T| + 2 \times |T| - 2 \times |ST|$, where $|T|$ denotes the number of test cases to be prioritized and $|ST|$ denotes the number of statements of the program. As it is usually costly to solve ILP problems, it may be necessary to further reduce the size of the ILP model through optimization.

The basic idea of the optimization is to reduce statements covered by exactly the same set of test cases into one super statement. For any given test suite $T$ and a program $P$, a super statement is formally defined as a set of statements that are executed by the same test cases in $T$. For any statement $s_{i}$ and $s_{j}$, if there exists a test case in $T$ that executes one but only one of these two statements, the two statements must belong to different super statements. Based on the concept of super statements, the statements of $P$ may be divided into several sets of statements, each of which is a super statement. Moreover, these super statements have no common statements. Let us use the example in Table 1 to explain the concept of super statements. The five test cases (including A, B, C, D, and E) divide the nine statements into the following eight super statements: $\{s_{1}\}, \{s_{2}\}, \{s_{3},s_{6}\}, \{s_{4}\}, \{s_{5}\}, \{s_{7}\}, \{s_{8}\}$, and $\{s_{9}\}$.
Note that the notion of super statements is similar to that of blocks. However, statements belonging to one block also belong to one super statement, but statements from one super statement may belong to more than one block. In particular, one super statement (which represents the set of statements executed by the same set of test cases) can be denoted as a tuple \((sst_k, num_k)\), where \(sst_k\) (1 \(\leq k \leq m\)) denotes the \(k\)-th super statement and \(num_k\) (1 \(\leq num_k \leq m\)) denotes that the \(k\)-th super statement actually contains \(num_k\) statements. Thus, program \(ST\) can be denoted as a set of super statements: \(ST = \{(sst_1, num_1), (sst_2, num_2), \ldots, (sst_m', num_{m'})\}\), where 1 \(\leq m' \leq m\) and \(\sum_{k=1}^{m'} num_k = m\). That is, the eight super statements in Table 1 can be denoted as \(ST = \{(sst_1, 1), (sst_2, 1), (sst_3, 2), (sst_4, 1), (sst_5, 1), (sst_6, 1), (sst_7, 1), (sst_8, 1)\}\).

Based on the preceding representation of the program \(ST\), decision variables in Formula 4 can be redefined on the basis of super statements. That is to say, \(y_{jk}\) (1 \(\leq j \leq n\) and 1 \(\leq k \leq m'\)) is 1 if and only if the first \(j\) test cases in \(T'\) covers the super statement \((sst_k, num_k)\). Furthermore, \(m\) in Formulas 7, 8, 9, 10, and 11 should be replaced by \(m'\). The object function should be redefined as Formula 14, as the number of statements in each super statement also impacts the APSC value.

\[
\text{maximize} \sum_{j=1}^{n-1} \sum_{k=1}^{m'} y_{jk} \times \text{num}_k \tag{14}
\]

Utilizing the notion of super statements rather than statements, the number of variables and the number of constraints decrease so that it becomes less costly to solve the ILP model for optimal test-case prioritization. On the other side, the optimization process does not change the functionality of the ILP model presented from Section 3.1 to Section 3.3 because super statements can be viewed as another way to represent statements. That is, the optimization introduced in this subsection changes only representation of the preceding ILP model. For example, using the statement coverage directly, we have the following 45 variables \(c_{11}, c_{12}, \ldots, c_{19}, \ldots, c_{51}, c_{52}, \ldots, c_{59}\), each of whose values represents whether the corresponding test case in Table 1 covers the corresponding statement. For statements \(s_3\) and \(s_6\), they are covered by the same test cases (i.e., \(A, B,\) and \(D\)), and thus \(c_{53}\) is always equal to \(c_{56}\) where \(i = 1 \ldots 5\). Based on the observation that 
\(c_{53} = c_{55}\), we transform the ILP model by using the notion of super statement. In particular, we use the following 40 variables for statement coverage, which are \(c_{11}, c_{12}, \ldots, c_{18}, \ldots, c_{51}, c_{52}, \ldots, c_{58}\), each of whose values represents whether the corresponding test case covers the corresponding super statement and statements \(s_3\) and \(s_6\) belong to the same super statement. That is, using the notion of super statements, some statements are combined into one (which is actually super statement) because test cases in \(T\) cannot distinguish them. Moreover, the optimal technique based on other structural coverage (e.g., method/function coverage) can be optimized in the same way. Furthermore, this notion can also be used to improve existing test-case prioritization techniques.

4 Study I

In this study, we investigate empirical properties of optimal coverage-based test-case prioritization by answering the following research questions.

- **RQ1**: How do optimal coverage-based test-case prioritization and additional coverage-based test-case prioritization perform differently in terms of structural coverage (using the APxC metric)?
- **RQ2**: How do optimal coverage-based test-case prioritization and additional coverage-based test-case prioritization perform differently in terms of fault detection (using the APFD metric)?
- **RQ3**: What is the difference in execution time between optimal coverage-based test-case prioritization and additional coverage-based test-case prioritization?

The first research question intends to investigate whether the optimal technique can be superior to the additional technique in terms of
TABLE 2
Studied Programs

<table>
<thead>
<tr>
<th>Abb.</th>
<th>Program</th>
<th>LOC</th>
<th>#Method</th>
<th>#Groups of mutants</th>
<th>#Test</th>
</tr>
</thead>
<tbody>
<tr>
<td>pt</td>
<td>print_tokens</td>
<td>203</td>
<td>20</td>
<td>2,356</td>
<td>4,072</td>
</tr>
<tr>
<td>pt2</td>
<td>print_tokens2</td>
<td>203</td>
<td>21</td>
<td>2,036</td>
<td>4,057</td>
</tr>
<tr>
<td>rep</td>
<td>replace</td>
<td>289</td>
<td>23</td>
<td>760</td>
<td>5,542</td>
</tr>
<tr>
<td>sch</td>
<td>schedule</td>
<td>162</td>
<td>20</td>
<td>829</td>
<td>2,627</td>
</tr>
<tr>
<td>sch2</td>
<td>schedule2</td>
<td>144</td>
<td>18</td>
<td>1,310</td>
<td>2,683</td>
</tr>
<tr>
<td>tca</td>
<td>tcas</td>
<td>67</td>
<td>11</td>
<td>967</td>
<td>1,592</td>
</tr>
<tr>
<td>tot</td>
<td>tlot_info</td>
<td>136</td>
<td>9</td>
<td>1,790</td>
<td>1,028</td>
</tr>
<tr>
<td>sp</td>
<td>space</td>
<td>6,218</td>
<td>136</td>
<td>9,657</td>
<td>13,585</td>
</tr>
<tr>
<td>jt</td>
<td>jtopas-v0</td>
<td>1,831</td>
<td>267</td>
<td>56</td>
<td>95</td>
</tr>
<tr>
<td>jt</td>
<td>jtopas-v1</td>
<td>1,871</td>
<td>269</td>
<td>63</td>
<td>126</td>
</tr>
<tr>
<td></td>
<td>jtopas-v2</td>
<td>1,912</td>
<td>274</td>
<td>84</td>
<td>128</td>
</tr>
<tr>
<td>sie</td>
<td>siena-v0</td>
<td>2,225</td>
<td>196</td>
<td>50</td>
<td>567</td>
</tr>
<tr>
<td>sie</td>
<td>siena-v1</td>
<td>2,206</td>
<td>187</td>
<td>53</td>
<td>567</td>
</tr>
<tr>
<td></td>
<td>siena-v2</td>
<td>2,207</td>
<td>187</td>
<td>53</td>
<td>567</td>
</tr>
<tr>
<td>sie</td>
<td>siena-v3</td>
<td>2,242</td>
<td>197</td>
<td>54</td>
<td>567</td>
</tr>
<tr>
<td></td>
<td>siena-v4</td>
<td>2,253</td>
<td>198</td>
<td>57</td>
<td>567</td>
</tr>
<tr>
<td></td>
<td>siena-v5</td>
<td>2,255</td>
<td>198</td>
<td>57</td>
<td>567</td>
</tr>
<tr>
<td>sie</td>
<td>siena-v6</td>
<td>2,246</td>
<td>198</td>
<td>57</td>
<td>567</td>
</tr>
<tr>
<td></td>
<td>siena-v7</td>
<td>2,232</td>
<td>194</td>
<td>59</td>
<td>567</td>
</tr>
</tbody>
</table>

Note: The second research question intends to investigate whether the optimal technique can be superior to the additional technique in terms of fault detection. The third research question intends to investigate how much more cost in terms of execution time the optimal technique has than the additional technique.

In the remaining of this section, we present the independent and dependent variables in Sections 4.1 and 4.2, the target programs, faults, and test suites used in our experiment in Section 4.3, the experimental procedure in Section 4.4, the threats to validity in Section 4.5, the results in Section 4.6, and the findings of our empirical study in Section 4.7.

4.1 Independent Variables

This empirical study focuses on comparing optimal coverage-based test-case prioritization and additional coverage-based test-case prioritization. To control the impact of coverage criteria, we consider two intensively investigated coverage criteria: statement coverage and method (function) coverage. Therefore, our empirical study uses the following four independent variables: (1) the optimal test-case prioritization technique based on statement coverage, (2) the optimal test-case prioritization technique based on method (function) coverage, (3) the additional test-case prioritization technique based on statement coverage, and (4) the additional test-case prioritization technique based on method (function) coverage.

Note that Section 3 presents only the optimal test-case prioritization technique using statement coverage. However, it is straightforward to extend this technique for test-case prioritization using method (function) coverage. That is, the optimal test-case prioritization technique based on either statement coverage or method (function) coverage is implemented following the same ways as described in Section 3, whose difference lies in only the coverage granularity.

Furthermore, in the test-case prioritization process of the additional technique, it is possible that some earlier selected test cases of the test suite \( T \) have achieved the same statement coverage as the whole test suite. In such scenario, we repeat the prioritization strategy by assuming that no test cases have been selected and scheduling the remaining unselected test cases. That is, we implement the additional test-case prioritization technique by applying the additional strategy again to the remaining unselected test cases. Note that we repeat the additional greedy strategy more than once because prior work on test case prioritization [2], [16], [17], [15] usually implements the additional technique in the same way.

4.2 Dependent Variables

According to our research questions, we consider the following three dependent variables.

- **Values for the APxC metric.** The APxC metric (defined in Formula 2) is to measure the effectiveness of the preceding four techniques for structural coverage. Note that the APxC metric becomes APSC for statement coverage and APMC for method (function) coverage.

- **Values for the APFD metric.** The APFD metric (defined in Formula 1) is to measure the effectiveness of the preceding four techniques for fault detection.

- **Execution time.** The execution time is for measuring the time efficiency of the preceding four techniques.

---

5. Whether the additional greedy strategy repeats more than once in the implementation of the additional technique, does not affect APxC values, but may affect APFD values.
4.3 Programs, Faults, and Test Suites

In this study, we used eight C programs and eleven versions of two Java programs that have been widely used in previous studies of test-case prioritization [2], [4]. These target programs are all available from the Software-Infrastructure Repository (SIR\(^6\)). Table 2 presents the statistics of these target programs, where the last two columns present the number of mutant groups (each mutant group consists of five mutants) injected into the target programs and the total number of test cases in the test pool. The former eight programs are written in C, whereas the latter programs are written in Java.

As SIR provides a large number of test cases in the test pool for each C program, in our empirical study, we created a series of test suites for each C program based on random selection using a strategy similar to Rothermel et al. [6], Elbaum et al. [1], [2], and Li et al. [4]. In particular, for each program, we created 100 test suites each containing 10 test cases, 100 test suites each containing 20 test cases, 100 test suites each containing 30 test cases, and 100 test suites each containing 40 test cases. When creating a test suite, we employed one or more rounds. The aim of using more than one round in our study was to control the sizes of created test suites. In one round, we adopted the same strategy used by Rothermel et al. [6], Elbaum et al. [1], [2], and Li et al. [4], which is to repeatedly select one test case randomly that can increase statement coverage of previously selected test cases in this round. Given a number \(n\) (\(n=10, 20, 30, \) or 40), while the total number of selected test cases is smaller than \(n\), we continuously employed another round. When the total number of selected test cases is larger than \(n\), we stopped and used the first \(n\) selected test cases to form the test suite. As the Java programs in our empirical study do not have a large number of test cases available at SIR considering the scale of these Java programs, similar to prior studies on these Java programs, we used all the test cases of each Java program as a test suite.

As these target programs have only a small number of manually injected faults available in SIR, we generated faulty versions for each program using program mutation, following the procedure similar to prior research [18]. In particular, for each program, we used mutation testing tools (i.e., MuJava [19] for Java programs, and Proteum [20] for C programs using their default setting) to generate all mutants, randomly selected five unselected mutants and constructed a faulty version by grouping these mutants. Following this procedure, we constructed a lot of mutant groups, each of which is viewed as a program with multiple faults. As some mutants cannot be killed\(^7\) by any test cases of a test suite, we view these mutants as unqualified mutants\(^8\). As each mutant group consists of five randomly selected mutants, each group in our empirical study consists of 1-5 qualified mutants by removing those unqualified mutants. Furthermore, we excluded such pair of a test suite (denoted as \(T\)) and a mutant group (denoted as \(M\)) that any test case in \(T\) cannot kill any mutant in \(M\), because such a pair would always produce 0 APFD values for any execution order of the test cases and cannot be used in the comparison of any test-case prioritization techniques. Table 3 summarizes the number of used mutant groups containing various numbers of qualified mutants. According to this table, for each program, many of its mutant groups consist of more than one fault.

4.4 Experimental Procedure

For each program, we executed the program with a constructed test suite, recording the coverage information (i.e., which statements and methods have been executed by each test case). Based on the coverage information, we applied the optimal test-case prioritization techniques and the additional test-case prioritization techniques, recording the prioritized test cases. In particular, we used IBM’s integer linear pro-

---


7. If a test case reveals the fault injected by mutation, the test case is said to kill the corresponding mutant [21].

8. Among these unqualified mutants, some are equivalent mutants that always behave as the original program, whereas some are not equivalent mutants but they behave as the original program for the given test suite.
TABLE 3
Statistics on Faults

<table>
<thead>
<tr>
<th>Program</th>
<th>Number of qualified mutants</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>One</td>
</tr>
<tr>
<td>print_tokens</td>
<td>1,9680</td>
</tr>
<tr>
<td>print_tokens2</td>
<td>3,928</td>
</tr>
<tr>
<td>replace</td>
<td>51,726</td>
</tr>
<tr>
<td>schedule</td>
<td>757</td>
</tr>
<tr>
<td>schedule2</td>
<td>14,088</td>
</tr>
<tr>
<td>fcas</td>
<td>30,661</td>
</tr>
<tr>
<td>tot info</td>
<td>1,813</td>
</tr>
<tr>
<td>space</td>
<td>547</td>
</tr>
<tr>
<td>jtopas-v0</td>
<td>18</td>
</tr>
<tr>
<td>jtopas-v1</td>
<td>13</td>
</tr>
<tr>
<td>jtopas-v2</td>
<td>10</td>
</tr>
<tr>
<td>siena-v0</td>
<td>0</td>
</tr>
<tr>
<td>siena-v1</td>
<td>0</td>
</tr>
<tr>
<td>siena-v2</td>
<td>0</td>
</tr>
<tr>
<td>siena-v3</td>
<td>0</td>
</tr>
<tr>
<td>siena-v4</td>
<td>0</td>
</tr>
<tr>
<td>siena-v5</td>
<td>0</td>
</tr>
<tr>
<td>siena-v6</td>
<td>1</td>
</tr>
<tr>
<td>siena-v7</td>
<td>0</td>
</tr>
</tbody>
</table>

4.5 Threats to Validity

The threat to internal validity lies in the implementation of the optimal test-case prioritization technique. To reduce this threat, we used IBM’s ILOG CPLEX in implementing the optimal test-case prioritization techniques. Moreover, to speedup test-case prioritization and reduce the cost in solving the ILP model, we use the notion of super statements in implementing the optimal test-case prioritization techniques and the additional test-case prioritization techniques. That is, for test-case prioritization based on statement coverage, including the optimal techniques and the additional techniques, we use the notion of super statements to represent statement coverage and similar super methods to represent method coverage.

Then we calculated the APSC values for the prioritized test suite of each test-case prioritization technique based on statement coverage, and the APMC values for the prioritized test suite of each test-case prioritization technique based on method (function) coverage. Furthermore, we calculated the APFD value of each prioritized test suite for each mutant group.

All the first study was conducted on a PC with an Intel E5530 16-Core Processor 2.40GHz, and the operating system is Ubuntu 9.10.

4.6 Results

To answer the three research questions, we use three subsections (each corresponding to one research question) to present the results of the first study[^10].

4.6.1 RQ1: Results on APxC Metric

Figures 1 and 2 present the column graphs of the mean APxC values (including APMC and APSC) of the optimal coverage-based techniques and the additional coverage-based techniques using method (function) coverage and statement coverage. Table 5 shows the standard

[^10]: The complete experimental data (including the implementation of the optimal technique and the additional technique) are accessible at https://github.com/ybwang1989/On-Optimal-Coverage-Based-Test-Case-Prioritization/wiki.
deviations of the APxC values, where the optimal technique is abbreviated as “Opt.” and the additional technique is abbreviated as “Add.”. To allow us to perform statistical analysis, similar to previous work [15], we group the results of the three versions of \textit{jtopas}. That is, although \textit{jtopas} has three versions \textit{jtopas}–v0, \textit{jtopas}–v1, and \textit{jtopas} – v2, we put all the APxC results of the three versions into one group without considering the difference between these versions and take all these APxC results as the results for \textit{jtopas}. Similarly, we do this kind of grouping when presenting other results (i.e., APFD and execution time) of \textit{jtopas} and \textit{siena}. From Figures 1 and 2, we make the following observations.

First, for each program, the mean APxC value of the additional technique is no larger than that of the optimal technique, considering either statement coverage or method (function) coverage. This observation is as expected since the optimal technique is designed to maximize the APxC values. However, except for a few programs (e.g., \textit{space} and \textit{jtopas}), the mean of APxC values of the additional technique are actually the same as those of the optimal technique. Moreover, even when the optimal technique outperforms the additional technique, their differences on the mean APxC results are ignorable. Second, the APSC values on statement coverage are smaller than the APMC values on method coverage. This observation is also as expected because it is easier to produce a prioritized test suite with high coarse-granularity coverage (e.g., method coverage) than with high fine-granularity coverage (e.g., statement coverage). Third, we compare the standard deviations of the APxC values and find that there are no significant differences between the two techniques for either statement coverage or method (function) coverage.

Furthermore, for each program, we calculated the percentage of cases on comparing the APxC values of the optimal technique and the APxC values of the additional technique. In particular, Opt. > Add. denotes the cases that the optimal technique outperforms the additional technique, Opt. = Add. denotes the cases that the optimal technique and the additional technique produce exactly the same APxC results, and Opt. < Add. denotes the cases that the additional technique outperforms the optimal technique. Figures 3 and 4 summary these results.

From Figures 3 and 4, the number of cases for Opt. < Add. is always 0, and the number of cases for Opt. = Add. is typically larger than that for Opt. > Add. (except for \textit{space} and \textit{jtopas}). This observation again demonstrates that the differences in APxC values between the two techniques are ignorable.
We further performed a one-way ANOVA analysis on the APxC values. The analysis results are shown by Table 4, where the significance level $\alpha$ is set to be 0.05. Moreover, the last row shows whether the hypothesis that there is no significant difference between the compared two techniques is rejected (denoted as R) or not rejected (denoted as N). According to this table, for either program and either coverage, the p-values are close to 1 and thus there are no significant differences between the two techniques. Furthermore, the F-values on method (function) coverage are no larger than those on statement coverage and the p-values on method coverage are no smaller than those on method coverage, demonstrating that the differences on method (function) coverage may be even smaller than those on statement coverage.

Note that for jtopas the optimal techniques seem better than the additional techniques from Figure 3, Figure 4, and Table 5. However, from Table 4 there is no significant difference between the two techniques because jtopas has only three APSC results and three APMC results.

### 4.6.2 RQ2: Results on APFD Metric

Figures 5 and 6 present the column graphs of the mean of the APFD values of the optimal coverage-based technique and the additional coverage-based technique using method (function) coverage and statement coverage. Table 6 presents the standard deviations of these APFD values. Unlike the APxC values given by Figure 1, Figure 2, and Table 5, Figure 5, Figure 6, and Table 6 demonstrate that, for each program, the mean APFD value of the additional technique is larger than that of the optimal technique, whereas the standard deviations on the APFD values also indicate the superiority of the additional technique.

Figures 7 and 8 present the statistics on cases where the additional technique outperforms the optimal technique, is equal to the optimal technique, and is inferior to the optimal technique, respectively. These two figures can confirm the superiority of the additional technique in general.

To check whether the differences are significant, we also perform a one-way ANOVA analysis for the APFD value, where significance level is also set to be 0.05. The analysis results are shown by Table 7. From this table, the differences between the two techniques are usually significant, except for the two Java programs. Note that the number of pairs of test suites and mutant groups for the Java programs are much smaller than that for the

---

11. The one-way ANOVA analysis was performed using SPSS 15.0, which is an analytical software and accessible at http://www.spss.com.
TABLE 4
ANOVA Analysis on APxC Results between the Optimal Technique and the Additional Technique

<table>
<thead>
<tr>
<th>Test-case prioritization on method coverage</th>
<th>Test-case prioritization on statement coverage</th>
</tr>
</thead>
<tbody>
<tr>
<td>pt</td>
<td>pt2</td>
</tr>
<tr>
<td>----</td>
<td>-----</td>
</tr>
<tr>
<td>F-value</td>
<td>0.00</td>
</tr>
<tr>
<td>p-value</td>
<td>1.00</td>
</tr>
<tr>
<td>Result</td>
<td>N</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>pt</th>
<th>pt2</th>
<th>rep</th>
<th>sch</th>
<th>sch2</th>
<th>tca</th>
<th>tot</th>
<th>sp</th>
<th>jt</th>
<th>sie</th>
</tr>
</thead>
<tbody>
<tr>
<td>F-value</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
</tr>
<tr>
<td>p-value</td>
<td>0.96</td>
<td>0.92</td>
<td>0.88</td>
<td>0.98</td>
<td>1.00</td>
<td>1.00</td>
<td>0.95</td>
<td>0.84</td>
<td>0.94</td>
</tr>
<tr>
<td>Result</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>N</td>
</tr>
</tbody>
</table>

TABLE 5
Standard Deviations of APxC Results

<table>
<thead>
<tr>
<th>Test-case prioritization on method coverage</th>
<th>Test-case prioritization on statement coverage</th>
</tr>
</thead>
<tbody>
<tr>
<td>Opt.</td>
<td>0.02</td>
</tr>
<tr>
<td>Add.</td>
<td>0.02</td>
</tr>
<tr>
<td>Opt.</td>
<td>0.01</td>
</tr>
<tr>
<td>Add.</td>
<td>0.01</td>
</tr>
</tbody>
</table>

TABLE 6
Standard Deviations of APFD Results

<table>
<thead>
<tr>
<th>Test-case prioritization on method coverage</th>
<th>Test-case prioritization on statement coverage</th>
</tr>
</thead>
<tbody>
<tr>
<td>Opt.</td>
<td>0.10</td>
</tr>
<tr>
<td>Add.</td>
<td>0.09</td>
</tr>
<tr>
<td>Opt.</td>
<td>0.11</td>
</tr>
<tr>
<td>Add.</td>
<td>0.10</td>
</tr>
</tbody>
</table>

TABLE 7
ANOVA Analysis on APFD Results between the Optimal Technique and the Additional Technique

<table>
<thead>
<tr>
<th>Test-case prioritization on method coverage</th>
</tr>
</thead>
<tbody>
<tr>
<td>pt</td>
</tr>
<tr>
<td>----</td>
</tr>
<tr>
<td>F-value</td>
</tr>
<tr>
<td>p-value</td>
</tr>
<tr>
<td>Result</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Test-case prioritization on statement coverage</th>
</tr>
</thead>
<tbody>
<tr>
<td>pt</td>
</tr>
<tr>
<td>----</td>
</tr>
<tr>
<td>F-value</td>
</tr>
<tr>
<td>p-value</td>
</tr>
<tr>
<td>Result</td>
</tr>
</tbody>
</table>

Fig. 7. Comparison on APFD Results between the Optimal Technique and the Additional Technique on Method Coverage

Fig. 8. Comparison on APFD Results between the Optimal Technique and the Additional Technique on Statement Coverage

4.6.3 RQ3: Results on Execution Time

Table 8 presents the average execution time (in seconds) of the optimal technique and the additional technique for each program using either method (function) coverage and statement coverage for C programs. The insignificance may be due to the insufficient sampling space.
coverage\textsuperscript{12}. As the execution time consists of only the time used in test-case prioritization, the execution time is dependent on the number of prioritized test cases and thus for each program we list the prioritization time for the test suites of the same size. In particular, we use 10, 20, 30, and 40 to represent test suites with the corresponding sizes. As different versions of each Java program are of similar sizes and have test suites of similar sizes, we group results on these versions into one row. Note that each version of Java programs has only one test suite.

From Table 8, we can observe that the prioritization time of the optimal technique is typically tens of or even hundreds of more times than that of the additional technique, although the prioritization time of the optimal technique is still acceptable.

<table>
<thead>
<tr>
<th>Tech</th>
<th>Coverage</th>
<th>Method Coverage</th>
<th>Statement Coverage</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>10</td>
<td>20</td>
<td>30</td>
</tr>
<tr>
<td>pt1</td>
<td>Opt.</td>
<td>3.42</td>
<td>3.90</td>
</tr>
<tr>
<td></td>
<td>Add.</td>
<td>0.00</td>
<td>0.00</td>
</tr>
<tr>
<td>pt2</td>
<td>Opt.</td>
<td>3.29</td>
<td>3.74</td>
</tr>
<tr>
<td></td>
<td>Add.</td>
<td>0.00</td>
<td>0.00</td>
</tr>
<tr>
<td>rep</td>
<td>Opt.</td>
<td>3.26</td>
<td>3.68</td>
</tr>
<tr>
<td></td>
<td>Add.</td>
<td>0.00</td>
<td>0.00</td>
</tr>
<tr>
<td>sch</td>
<td>Opt.</td>
<td>3.29</td>
<td>3.62</td>
</tr>
<tr>
<td></td>
<td>Add.</td>
<td>0.00</td>
<td>0.00</td>
</tr>
<tr>
<td>sch2</td>
<td>Opt.</td>
<td>3.25</td>
<td>3.46</td>
</tr>
<tr>
<td></td>
<td>Add.</td>
<td>0.00</td>
<td>0.00</td>
</tr>
<tr>
<td>tca</td>
<td>Opt.</td>
<td>3.29</td>
<td>3.50</td>
</tr>
<tr>
<td></td>
<td>Add.</td>
<td>0.00</td>
<td>0.00</td>
</tr>
<tr>
<td>tot</td>
<td>Opt.</td>
<td>3.27</td>
<td>3.39</td>
</tr>
<tr>
<td></td>
<td>Add.</td>
<td>0.00</td>
<td>0.00</td>
</tr>
<tr>
<td>sp</td>
<td>Opt.</td>
<td>3.50</td>
<td>4.90</td>
</tr>
<tr>
<td></td>
<td>Add.</td>
<td>0.00</td>
<td>0.00</td>
</tr>
<tr>
<td>jt</td>
<td>Opt.</td>
<td>3.70</td>
<td>206.22</td>
</tr>
<tr>
<td></td>
<td>Add.</td>
<td>0.19</td>
<td>0.75</td>
</tr>
<tr>
<td>sie</td>
<td>Opt.</td>
<td>3.344.45</td>
<td>2,418.80</td>
</tr>
<tr>
<td></td>
<td>Add.</td>
<td>2.82</td>
<td>6.61</td>
</tr>
</tbody>
</table>

5 Study II

As optimal coverage-based test-case prioritization takes structural coverage as the intermediate goal and cannot guarantee to achieve the optimal APFD results, it is interesting to learn how good this technique is by comparing with the ideal order of test cases, which has the largest APFD results. Therefore, we conduct another study to investigate how good optimal

4.7 Summary

We summarize the main findings of the first empirical study as follows. First, the optimal...
coverage-based test-case prioritization and additional coverage-based test-case prioritization are by comparing with the ideal optimal test-case prioritization.

To answer this research question, we implemented the ideal optimal test-case prioritization technique, which schedules the execution order of test cases based on their detected faults. This ideal optimal test-case prioritization technique serves as the control technique to show the upper bound of test-case prioritization, because this technique assumes that it is known which faults each test case detects and is not applicable in practice.

5.1 Setup

In this study, we used six versions of two C projects and seven versions of three Java projects that have also been used in previous studies of test-case prioritization [23]. Table 9 presents the statistics of these programs. The first six programs are written in C, whereas the latter seven programs are written in Java. Following the same procedure as Section 4, we constructed faulty versions for C programs by grouping mutants generated by a mutation testing tool MutGen [24] and constructed faulty versions for Java programs by grouping mutants generated by a mutation testing tool Javalanche [25]. Table 10 summarizes the number of used mutant groups containing various numbers of qualified mutants.

Similar to the first study in Section 4, for each program, we constructed test suites for C programs and used the existing test suites for Java programs. Based on the coverage information of each test suite, we applied the optimal test-case prioritization technique implemented using GUIB13 and the additional test-case prioritization technique, recording the prioritized test cases.

To learn the upper bound of test-case prioritization, we implemented the ideal optimal test-case prioritization technique as a control technique. To generate the optimal order of test cases, it is necessary to consider all possible orders, and thus the worst-case runtime of the ideal optimal algorithm may be exponential in test suite size [6]. Therefore, Rothermel et al. [6] proposed to use an additional greedy strategy instead and called it an optimal technique, although the additional greedy strategy may not always produce the optimal order. In this article, we follow this work and implement the ideal optimal technique in the same way [6]. In particular, for each program with multiple faults (i.e., a mutant group), we prioritized test cases based on their number of detected faults that have been not detected by existing selected test cases. This ideal technique produces an ideal execution order of test cases. However, this ideal optimal technique is not practical because whether a fault is detected by a test case is not known before testing.

Then we calculated the APSC values for the prioritized test suite of each test-case prioritization technique based on statement coverage, and the APMC values for the prioritized test suite of each test-case prioritization technique based on method (function) coverage. Furthermore, we calculated the APFD value of each prioritized test suite for each mutant group. All this study was conducted on a PC with a Core i7-3770 Processor 3.4GHz whose operating system is Windows 7.

5.2 Results

The optimal technique proposed in this article takes the intermediate goal (i.e., structural coverage) as a goal in scheduling test cases,
TABLE 9
Studied Programs

<table>
<thead>
<tr>
<th>Abb.</th>
<th>Program</th>
<th>LOC</th>
<th>#Method</th>
<th>#Groups of mutants</th>
<th>#Test</th>
</tr>
</thead>
<tbody>
<tr>
<td>gz</td>
<td>gzip-v0</td>
<td>7,050</td>
<td>89</td>
<td>1,997</td>
<td>214</td>
</tr>
<tr>
<td></td>
<td>gzip-v1</td>
<td>7,266</td>
<td>88</td>
<td>2,004</td>
<td>214</td>
</tr>
<tr>
<td></td>
<td>gzip-v2</td>
<td>7,975</td>
<td>104</td>
<td>2,276</td>
<td>214</td>
</tr>
<tr>
<td>gp</td>
<td>grep-v0</td>
<td>10,929</td>
<td>133</td>
<td>11,114</td>
<td>470</td>
</tr>
<tr>
<td></td>
<td>grep-v1</td>
<td>12,555</td>
<td>149</td>
<td>12,472</td>
<td>470</td>
</tr>
<tr>
<td></td>
<td>grep-v2</td>
<td>13,128</td>
<td>155</td>
<td>12,959</td>
<td>470</td>
</tr>
<tr>
<td>xml</td>
<td>xmlsecurity-v0</td>
<td>35,579</td>
<td>1,231</td>
<td>810</td>
<td>97</td>
</tr>
<tr>
<td></td>
<td>xmlsecurity-v1</td>
<td>35,622</td>
<td>1,234</td>
<td>810</td>
<td>97</td>
</tr>
<tr>
<td></td>
<td>xmlsecurity-v2</td>
<td>33,523</td>
<td>1,146</td>
<td>823</td>
<td>95</td>
</tr>
<tr>
<td>tm</td>
<td>time&amp;money-v0</td>
<td>1,589</td>
<td>212</td>
<td>189</td>
<td>104</td>
</tr>
<tr>
<td></td>
<td>time&amp;money-v1</td>
<td>2,124</td>
<td>212</td>
<td>278</td>
<td>146</td>
</tr>
<tr>
<td>jg</td>
<td>jgrapht-v0</td>
<td>8,605</td>
<td>310</td>
<td>194</td>
<td>48</td>
</tr>
<tr>
<td></td>
<td>jgrapht-v1</td>
<td>11,251</td>
<td>377</td>
<td>292</td>
<td>63</td>
</tr>
</tbody>
</table>

whereas the ideal optimal technique takes the ultimate goal (i.e., detected faults) as a goal in scheduling test cases. Thus, the ideal optimal technique is expected to produce an ideal execution order of test cases, although the ideal technique is not practical. To learn the difference between the optimal technique and the ideal technique, we perform a one-way ANOVA analysis for the APxC results and the APFD results between the optimal technique and the ideal optimal technique, whose analysis results are shown by Table 11 and Table 12.

As there are a huge number of APFD results (over 1,000,000) for each C subject, we cannot conduct the one-way ANOVA analysis using SPSS on our PC. Therefore, for each C program (i.e., gzip-v0, gzip-v1, gzip-v2, grep-v0, grep-v1, and grep-v2), we conducted the one-way ANOVA analysis on the results of all test suites consisting of the same number of test cases separately, and recorded their analysis results. Then for each C subject (i.e., gzip and grep) we listed the range of their analysis results in Table 12. From this table, the optimal technique significantly outperforms the ideal technique in terms of APxC results, but the latter significantly outperforms the former in terms of APFD results.

Similarly, we perform a one-way ANOVA analysis for the APxC results and the APFD results between the additional technique and the ideal technique, whose analysis results are shown by Table 13 and Table 14. From this table, we got the similar conclusion as the comparison between the optimal technique and the ideal technique. That is, the additional technique significantly outperforms the ideal technique in terms of APxC results, but the latter significantly outperforms the former in terms of APFD results.

From this study, the ideal optimal test-case prioritization achieves the best APFD results but the worst APxC results. This observation also consolidates our findings in the first study. That is, it is not worthwhile to pursue optimality

TABLE 11
ANOVA Analysis on APxC Results between the Optimal Technique and the Ideal Technique

<table>
<thead>
<tr>
<th>Test-case prioritization on method coverage</th>
<th>gz</th>
<th>gp</th>
<th>xml</th>
<th>tm</th>
<th>jg</th>
</tr>
</thead>
<tbody>
<tr>
<td>F-value</td>
<td>12151.32</td>
<td>9721.26</td>
<td>19.81</td>
<td>456.15</td>
<td>7.51</td>
</tr>
<tr>
<td>p-value</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.01</td>
</tr>
<tr>
<td>Result</td>
<td>R</td>
<td>R</td>
<td>R</td>
<td>R</td>
<td>R</td>
</tr>
</tbody>
</table>

TABLE 12
ANOVA Analysis on APFD Results between the Optimal Technique and the Ideal Technique

<table>
<thead>
<tr>
<th>Test-case prioritization on method coverage</th>
<th>gz</th>
<th>gp</th>
<th>xml</th>
<th>tm</th>
<th>jg</th>
</tr>
</thead>
<tbody>
<tr>
<td>F-value</td>
<td>82.89-1111065.00</td>
<td>15477.13-31228.53</td>
<td>599.97</td>
<td>300.95</td>
<td>376.66</td>
</tr>
<tr>
<td>p-value</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
</tr>
<tr>
<td>Result</td>
<td>R</td>
<td>R</td>
<td>R</td>
<td>R</td>
<td>R</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Test-case prioritization on statement coverage</th>
<th>gz</th>
<th>gp</th>
<th>xml</th>
<th>tm</th>
<th>jg</th>
</tr>
</thead>
<tbody>
<tr>
<td>F-value</td>
<td>19306.51</td>
<td>7880.10</td>
<td>35.91</td>
<td>218.34</td>
<td>7.38</td>
</tr>
<tr>
<td>p-value</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.01</td>
</tr>
<tr>
<td>Result</td>
<td>R</td>
<td>R</td>
<td>R</td>
<td>R</td>
<td>R</td>
</tr>
</tbody>
</table>
### TABLE 13
ANOVA Analysis on APxC Results between the Additional Technique and the Ideal Technique

<table>
<thead>
<tr>
<th>Test-case prioritization on method coverage</th>
<th>gz</th>
<th>gp</th>
<th>xml</th>
<th>tm</th>
<th>Jg</th>
</tr>
</thead>
<tbody>
<tr>
<td>F-value</td>
<td>10944.93</td>
<td>9025.92</td>
<td>19.66</td>
<td>435.86</td>
<td>7.51</td>
</tr>
<tr>
<td>p-value</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.01</td>
</tr>
<tr>
<td>Result</td>
<td>R</td>
<td>R</td>
<td>R</td>
<td>R</td>
<td>R</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Test-case prioritization on statement coverage</th>
<th>gz</th>
<th>gp</th>
<th>xml</th>
<th>tm</th>
<th>Jg</th>
</tr>
</thead>
<tbody>
<tr>
<td>F-value</td>
<td>16341.50</td>
<td>6724.87</td>
<td>35.77</td>
<td>218.25</td>
<td>7.38</td>
</tr>
<tr>
<td>p-value</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.01</td>
</tr>
<tr>
<td>Result</td>
<td>R</td>
<td>R</td>
<td>R</td>
<td>R</td>
<td>R</td>
</tr>
</tbody>
</table>

### TABLE 14
ANOVA Analysis on APFD Results between the Additional Technique and the Ideal Technique

<table>
<thead>
<tr>
<th>Test-case prioritization on method coverage</th>
<th>gz</th>
<th>gp</th>
<th>xml</th>
<th>tm</th>
<th>Jg</th>
</tr>
</thead>
<tbody>
<tr>
<td>F-value</td>
<td>68.63-17492.00</td>
<td>24618.58-30140.31</td>
<td>600.42</td>
<td>253.92</td>
<td>438.16</td>
</tr>
<tr>
<td>p-value</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
</tr>
<tr>
<td>Result</td>
<td>R</td>
<td>R</td>
<td>R</td>
<td>R</td>
<td>R</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Test-case prioritization on statement coverage</th>
<th>gz</th>
<th>gp</th>
<th>xml</th>
<th>tm</th>
<th>Jg</th>
</tr>
</thead>
<tbody>
<tr>
<td>F-value</td>
<td>26.28-20885.99</td>
<td>13818.13-28622.88</td>
<td>525.88</td>
<td>263.44</td>
<td>382.03</td>
</tr>
<tr>
<td>p-value</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
</tr>
<tr>
<td>Result</td>
<td>R</td>
<td>R</td>
<td>R</td>
<td>R</td>
<td>R</td>
</tr>
</tbody>
</table>

by taking the coverage as an intermediate goal in test-case prioritization.

### 6 Discussion

In this section, we discuss some related issues.

First, due to the difference between the ultimate goal (i.e., APFD) and the intermediate goal (i.e., APxC), when pursing the maximum APxC values, the optimal coverage-based test-case prioritization technique generates some scheduled test suites that overfit for the APxC values and have smaller APFD values than the simple additional coverage-based test-case prioritization technique. As the fault-detection capability of test cases is unknown without running test cases, it is impossible to use the ultimate goal to guide test-case prioritization and thus it is necessary to use an intermediate goal instead. To bridge the gap between the ultimate goal and the intermediate goal and overcome the overfitting problem in optimal test-case prioritization, researchers may tune the intermediate goal slightly. For example, in our previous work [22], we unified the two typical greedy test-case prioritization techniques (i.e., the total test-case prioritization and the additional test-case prioritization techniques) by using a parameter $p$ on APxC.

Second, although our empirical study is on test-case prioritization, its conclusions may be generalized to other software-testing tasks [26, 27] (e.g., test-suite reduction [28]). Test-case prioritization is only one of the typical software-testing tasks with the characteristics that pursing the intermediate-goal optimality may become overfit for the intermediate goal and thus become less effective than some heuristic or greedy techniques for the ultimate goal. Other software-testing tasks share the same characteristics. Test-suite reduction is proposed in regression testing, whose aim is to find the minimal set of the existing test suite so that the minimal subset has the same fault-detection capability as the whole test suite. As it is impossible to know the fault-detection capability of test cases without running test cases, researchers in test-suite reduction also tend to use intermediate goals (e.g., structural coverage) to replace the ultimate goal (i.e., fault-detection capability). When pursing the optimality, existing test-suite reduction based on ultimate goals may also become less effective when measured on the ultimate goal. In this article, we have investigated this problem based on the empirical study of test-case prioritization due to effort limit; a similar problem occurs in many other software-testing tasks and the conclusions from the empirical study may be generalized. In other words, researchers should take precautions in pursuing optimal solutions using the intermediate goal.

### 7 Related Work

Test-case prioritization is an optimization problem, similar to test-suite reduction, which is conceptually related to the set cover problem and the hitting set problem. For ease of presentation, we classify existing research on test-case prioritization into four groups.

The first group focuses on investigating various coverage criteria for test-case prioritization [29]. Besides statement
coverage, researchers have also investigated function/method coverage [1], branch coverage [30], definition-use association coverage, modified condition/decision coverage [31], specification coverage [32], the ordered sequence of program entities [33]. Furthermore, Korel et al. [3], [34] proposed model-based test-case prioritization, which first maps the change on the source code to model elements and then schedules the order of test cases based on their relevance to these model elements. As our research focuses on prioritization strategies, we adopt two typical coverage criteria in our empirical study: statement coverage and function/method coverage. Different from these coverage-based techniques, Ledru et al. [35] do not assume the existence of code or specification and thus proposed a prioritization technique based on the string distances on the text of test cases, which may be useful for initial testing.

The second group focuses on strategies for test-case prioritization. Besides the additional strategy, researchers have also widely investigated the random strategy and the total strategy. Li et al. [4] investigated some optimization techniques based on genetic algorithms and hill climbing for test-case prioritization. Their APxC results show that the additional strategy outperforms both the strategy based on a genetic algorithm and the strategy based on hill climbing, but the differences are insignificant. Different from their work, in this article, we formalize optimal coverage-based test-case prioritization as an ILP problem. Furthermore, Li et al. [36] conducted a simulation experiment to study five search algorithms for test-case prioritization, including the total greedy algorithm, the additional greedy algorithm, the 2-optimal greedy algorithm, the hill climbing algorithm, and the genetic algorithm. Their results show that the additional greedy algorithm and the 2-optimal greedy algorithm are both preferable and outperform the others. Jiang et al. [10] investigated a random adaptive strategy for test-case prioritization. According to their results, the random adaptive strategy is less effective in terms of APFD but also less costly than the additional strategy. Hao et al. [37] combined the output of test cases to improve test-case prioritization and this technique sometimes outperforms the total and the additional strategies in terms of APFD. Zhang et al. [22], [23] investigated strategies with mixed flavor of the additional strategy and the total strategy. Their results demonstrate that there are mixed strategies that can significantly outperform both the additional strategy and the total strategy in terms of APFD.

Our research focuses on investigating empirical properties of the optimal strategy in terms of APxC. To our knowledge, our research is the first study on the optimal strategy.

The third group focuses on practical constraints (e.g., fault severity [38] and time budget [11]) in test-case prioritization. For example, as time constraints may strongly affect the behavior of test-case prioritization, Suri and Singhal [39] proposed an ant colony optimization based technique to prioritize test cases with time constraints. Do et al. [40] conducted a series of experiments to evaluate the effects of time constraints on the costs and benefits of test-case prioritization. Marijan et al. [41] proposed to use the test execution time as a constraint and scheduled the execution order of test cases based on historical failure data, test execution time, and domain-specific heuristics for industrial conferencing software in continuous regression testing. As the cost of test cases and severity of faults vary, Huang et al. [42] proposed a cost-cognizant technique using historical records from the latest regression testing, which determines the execution order with a genetic algorithm. Zhang et al. [43] proposed to prioritize test cases based on varying test requirement priorities and test case costs by adapting existing test-case prioritization techniques. Note that the ILP model proposed by Zhang et al. [11] is for selecting test cases under the time constraint, but not a prioritization strategy like the ILP model presented in this article. Our research reported in this article focuses on investigating the optimal strategy in the context without these practical constraints. However, it should be interesting to further investigate the optimal strategy in contexts with one or more practical constraints.

The fourth group focuses on empirical studies on test-case prioritization [1], [2], [6]. Exist-
ing empirical studies investigate various factors (e.g., programming languages [44], coverage granularity [2], fault type [18], [21]) that may influence the effectiveness of existing test-case prioritization techniques, whereas our empirical study investigates empirical differences between the additional strategy and the optimal strategy. However, it may also be interesting to further investigate the influencing factors for the optimal strategy.

Furthermore, some researchers apply test-case prioritization to solve other software-engineering problems, e.g., mutation testing, fault localization. Zhang et al. [45] proposed to prioritize test cases based on the order that a test case killing the mutant is run earlier to reduce the cost of mutation testing. Sánchez et al. [46] proposed to apply test-case prioritization to testing software product lines. Yoo et al. [47] proposed a cluster-based test-case prioritization technique, which first clusters test cases using the agglomerative hierarchical clustering technique and then prioritizes these clusters to reduce human involvement in test-case prioritization.

8 Conclusion

As the additional strategy for coverage-based test-case prioritization is surprisingly competitive according to some prior work, we are curious about how the optimal strategy for coverage-based test-case prioritization would perform differently from the additional strategy for coverage-based test-case prioritization. Therefore, we have conducted an empirical study on comparing these two strategies (using APxC, APFD, and execution time) with statement coverage and method coverage. According to our empirical results, the optimal strategy is only slightly more effective than the additional strategy with regard to APxC, but significantly less effective than the additional strategy with regard to both APFD and the execution time, demonstrating the inferiority of the optimal strategy in practice. Furthermore, we have implemented the ideal optimal strategy. Taking this strategy as the upper bound of test-case prioritization, we have conducted another empirical study on comparing the effectiveness of the optimal and the additional strategies. From this empirical study, the ideal strategy significantly outperforms the optimal and additional strategies in terms of APFD, but significantly worse in terms of APxC.

Therefore, when pursing the optimality in test-case prioritization using the intermediate goal (i.e., APxC), the produced solutions are overfit for this goal and less effective than solutions produced by the greedy technique in terms of the ultimate goal (i.e., APFD). This conclusion indicates that it may not be worthwhile to pursue the optimality in test-case prioritization by taking coverage as an intermediate goal considering the efforts. In future work, we plan to further investigate the optimality issue in other software-testing tasks (e.g., test-case generation or test-suite reduction).

Acknowledgments

This work is supported by the National 973 Program of China No. 2015CB352201, the National Natural Science Foundation of China under Grants No.61421091, 61225007, 61529201, 61522201, and 61272157. This work is also supported by the National Science Foundation of the United States under grants No. CCF-1349666, CCF-1409423, CNS-1434582, CCF-1434590, CCF-1434596, CNS-1439481, and CNS-1513939. Besides, we would like to thank Shanshan Hou, Xinyi Wan, and Chao Guo. They contributed to the early discussion of this work.

References


Dan Hao is an Associate professor at the School of Electronics Engineering and Computer Science, Peking University, P.R. China. She received her Ph.D. in Computer Science from Peking University in 2008, and the B.S. in Computer Science from the Harbin Institute of Technology in 2002. Her current research interests include software testing and debugging. She is a senior member of ACM.

Lu Zhang is a professor at the School of Electronics Engineering and Computer Science, Peking University, P.R. China. He received both PhD and BSc in Computer Science from Peking University in 2000 and 1995 respectively. He was a postdoctoral researcher in Oxford Brookes University and University of Liverpool, UK. He served on the program committees of many prestigious conferences, such as FSE, ISSTA, and ASE. He was a program co-chair of SCAM2008 and will be a program co-chair of ICSM17. He has been on the editorial boards of Journal of Software Maintenance and Evolution: Research and Practice and Software Testing, Verification and Reliability. His current research interests include software testing and analysis, program comprehension, software maintenance and evolution, software reuse and component-based software development, and service computing.

Lei Zang is a third-year Computer Science Master student from Software Engineering Institute, Peking University. He will graduate in the July of 2016 and become a software engineer. His current research interests include software testing and analysis.

Yanbo Wang is a PhD student in the Department of Regional Economics, School of Government, Peking University. He received his B.S. in software engineering from Huazhong University of Science and Technology, Wuhan, China, in 2012. His major interests are in the areas of complex system, complex dynamics and economic computation.

Xingxia Wu is a software engineer at a regional bank, China. She received both the BS and MS degrees in computer science from Peking University. Her research interests include software testing and analysis.
Tao Xie is an Associate Professor in the Department of Computer Science at the University of Illinois at Urbana-Champaign, USA. His research interests are software testing, program analysis, software analytics, software security, and educational software engineering. He is a senior member of IEEE.