Regression Test Case Prioritization by Code Combinations Coverage

Rubing Huang, Quanjun Zhang, Dave Towey,
Weifeng Sun, Jinfu Chen

University of Nottingham
UK | CHINA | MALAYSIA

Faculty of Science and Engineering, University of Nottingham Ningbo China, 199 Taikang East Road, Ningbo, 315100, Zhejiang, China.

First published 2020

University of Nottingham
UK | CHINA | MALAYSIA

Regression Test Case Prioritization by Code Combinations Coverage

Rubing Huang[a,*], Quanjun Zhang[b], Dave Towey[c], Weifeng Sun[b], Jinfu Chen[b]

[a]School of Computer Science and Communication Engineering, and Jiangsu Key Laboratory of Security Technology for Industrial Cyberspace, Jiangsu University, Zhenjiang, Jiangsu 212013, P.R. China
[b]School of Computer Science and Communication Engineering, Jiangsu University, Zhenjiang, Jiangsu 212013, P.R. China
[c]School of Computer Science, The University of Nottingham Ningbo China, Ningbo, Zhejiang 315100, P.R. China.

Abstract

Regression test case prioritization (RTCP) aims to improve the rate of fault detection by executing more important test cases as early as possible. Various RTCP techniques have been proposed based on different coverage criteria. Among them, a majority of techniques leverage code coverage information to guide the prioritization process, with code units being considered individually, and in isolation. In this paper, we propose a new coverage criterion, *code combinations coverage*, that combines the concepts of code coverage and combination coverage. We apply this coverage criterion to RTCP, as a new prioritization technique, *code combinations coverage based prioritization* (CCCP). We report on empirical studies conducted to compare the testing effectiveness and efficiency of CCCP with four popular RTCP techniques: *total*, *additional*, *adaptive random*, and *search-based* test prioritization. The experimental results show that even when the lowest combination strength is assigned, overall, the CCCP fault detection rates are greater than those of the other four prioritization techniques. The CCCP prioritization costs are also found to be comparable to the additional test prioritization technique. Moreover, our results also show that when the combination strength is increased, CCCP provides higher fault detection rates than the state-of-the-art, regardless of the levels of code coverage.

Keywords: Software testing, regression testing, test case prioritization, code combinations coverage

1. Introduction

Modern software systems continuously evolve due to the fixing of detected bugs, the adding of new functionalities, and the refactoring of system architecture. Regression testing is conducted to ensure that the changed source code does not introduce new defects. However, it can become expensive to run an entire regression test suite because its size naturally increases during software maintenance and evolution: In an industrial case reported by Rothermel et al. [1], for example, the execution time for running the entire test suite could become several weeks.

Regression test case prioritization (RTCP) has become one of the most effective approaches to reduce the overheads in regression testing [2, 3, 4, 5, 6]. RTCP techniques reorder the execution sequence of regression test cases, aiming to execute those test cases more likely to detect faults (according to some award function) as early as possible [7, 8, 9].

Traditional RTCP techniques [1, 10, 11] usually use code coverage criteria to guide the prioritization process. Intuitively speaking, a code coverage criterion indicates the percentage of some code units (e.g. statements) covered by a test case. The expectation is that test cases with higher code coverage value have a greater chance of detecting faults [12]. Because of this, a goal of maximizing code coverage has been incorporated into various RTCP techniques, including greedy strategies [1]. Given a coverage criterion (e.g., method, branch, or statement coverage), the *total* strategy selects the next test case with greatest absolute coverage, whereas the *additional* strategy selects the one with greatest coverage of code units not already covered by the prioritized test cases. Furthermore, Li et al. [2] proposed two search-based RTCP techniques (a hill-climbing strategy and a genetic strategy) to explore the

*Corresponding author.
Email addresses: rbhuang@ujs.edu.cn (Rubing Huang), 2211708038@stmail.ujs.edu.cn (Quanjun Zhang), dave.towey@nottingham.edu.cn (Dave Towey), 2211808031@stmail.ujs.edu.cn (Weifeng Sun), jinfuchen@ujs.edu.cn (Jinfu Chen)

search space (the set of all permutations of the test cases) to find a sequence with a better fault detection rate. Jiang et al. [3] investigated adaptive random techniques [13] to prioritize test cases using code coverage criteria. In an attempt to bridge the gap between the two greedy strategies, Zhang et al. [10] proposed a unified approach based on the fault detection probability for each test case (referred to as a p value).

In this paper, we propose a new coverage criterion, *code combinations coverage*, that combines the concepts of code coverage [12] and combination coverage [14]: Given a set of regression test cases T, each test case is first transferred to an equally-sized tuple. Each position in this tuple is a binary value representing whether the corresponding item (such as branch, statement, or method) is covered by this test case. In other words, T is represented by a set of abstract test cases with binary values T'. The code combinations coverage of T is measured by the traditional combination coverage of T'. We apply this new coverage criterion to RTCP, proposing a new prioritization technique: *code combinations coverage based prioritization* (CCCP).

We conducted empirical studies on 14 versions of four Java programs, and 30 versions of five real-world Unix utility programs. Our goal was to investigate the testing effectiveness and efficiency of CCCP compared with four widely-used RTCP techniques — *total*, *additional*, *adaptive random*, and *search-based* test prioritization. The results show that when the lowest combination strength is assigned, overall, our approach has better fault detection rates than the other four test prioritization techniques. It not only achieves comparable testing efficiency to *additional*, but also requires much less prioritization time than the *adaptive random* and *search-based* techniques. In addition, while the *code coverage* granularity does not impact on the testing effectiveness of CCCP, the *test case* granularity does significantly impact on it. Furthermore, when the combination strength is increased, CCCP provides better fault detection rates than all other RTCP techniques, regardless of the level of code coverage.

The main contributions of this paper are:

- We propose a new coverage criterion called *code combinations coverage* that combines the concepts of code coverage and combination coverage.
- We apply code combinations coverage to RTCP, leading to a new prioritization technique called *code combinations coverage based prioritization* (CCCP).
- We report on empirical studies conducted to investigate the test effectiveness and efficiency of CCCP compared to four widely-used prioritization techniques, and also analyze the impact of code coverage granularity and test case granularity on the effectiveness of CCCP.
- We provide some guidelines for how to choose the combination strength and code-coverage level for CCCP, under different testing scenarios.

The rest of this paper is organized as follows: Section 2 presents some background information. Section 3 introduces the proposed approach. Section 4 presents the research questions, and explains details of the empirical study. Section 5 provides the detailed results of the study and answers the research questions. Section 6 discusses some related work, and Section 7 concludes this paper, including highlighting some potential future work.

2. Background

In this section, we provide some background information about abstract test cases and test case prioritization.

2.1. Abstract Test Cases

For the system under test (SUT), there are some parameters $p_1, p_2, \cdots, p_k$ that may influence its performance, such as configuration options, components, and user inputs. Each parameter p_i can take some discrete values to form the set V_i, which is finite. By selecting a value for each parameter, its combination becomes an *abstract test case* [15].

Definition 1. ***Abstract Test Case:*** *An abstract test case is a discrete test case that can be represented by a k-tuple $(v_1, v_2, \cdots, v_k)$, where v_i $(1 \leq i \leq k)$ is a value of a parameter p_i from a finite set V_i (i.e., $v_i \in V_i$).*

Each abstract test case covers some λ-wise tuples (called *λ-wise parameter-value combinations* [16] or *λ-wise schemas* [14]), where $1 \leq \lambda \leq k$. For example, an abstract test case $tc = (1, 3, 5, 7)$ covers the six 2-wise parameter-value combinations $(1, 3)$, $(1, 5)$, $(1, 7)$, $(3, 5)$, $(3, 7)$, and $(5, 7)$; and also covers the four 3-wise parameter-value combinations $(1, 3, 5)$, $(1, 3, 7)$, $(1, 5, 7)$, and $(3, 5, 7)$. Intuitively speaking, when $\lambda = k$, a λ-wise parameter-value combination becomes an abstract test case.

For ease of description, we define a function $CombSet(tc, \lambda)$ that returns a set of all λ-wise

parameter-value combinations covered by an abstract test case $tc = (v_1, v_2, \cdots, v_k)$, i.e.,

$$CombSet(tc, \lambda) = \left\{ (v_{i_1}, v_{i_2}, \cdots, v_{i_\lambda}) \middle| 1 \le i_1 < i_2 < \cdots < v_\lambda \le k \right\} \quad (1)$$

Obviously, the size of $CombSet(tc, \lambda)$ is equal to $C(k, \lambda)$ (the number of λ-combinations from k elements). To calculate the λ-wise parameter-value combinations covered by the set T of abstract test cases, the function $CombSet(T, \lambda)$ is defined as:

$$CombSet(T, \lambda) = \bigcup_{tc \in T} CombSet(tc, \lambda) \quad (2)$$

2.2. Test Case Prioritization

Regression test case prioritization (RTCP) [1] aims to reorder the test cases to realize a certain goal, such as exposing faults earlier. RTCP is formally defined as:

Definition 2. ***Regression Test Case Prioritization:*** *Given a regression test suite T, PT is the set of its all possible permutations, and f is an object function from PT to real numbers. The problem of RTCP [1] is to find $P' \in PT$, such that $\forall P'', P'' \in PT(P'' \neq P'), f(P') \geq f(P'')$.*

RTCP is an effective means to reduce the cost of regression testing, and has been widely investigated [1, 2, 3, 9], with a large number of studies focusing on the coverage criterion and prioritization algorithms. Intuitively, code coverage criteria can be regarded as characteristics of the test cases, and many prioritization algorithms have used coverage criteria to guide the prioritization process (such as the greedy strategies [1], search-based strategies [2], and adaptive random strategies [3]).

3. Approach

In this section, we introduce the details of test case prioritization by code combinations coverage.

3.1. Greedy Techniques

There are two widely investigated RTCP strategies: the *total* greedy strategy and the *additional* greedy strategy. The *total* strategy selects test cases according to a descending order of code units covered by the test case. The *additional* strategy also selects test cases according to a descending order, but uses the number of code units not already covered by previously selected test cases. According to previous studies [17, 18, 19], although seemingly simple, the greedy strategies (especially *additional*) perform better than most other RTCP techniques in terms of the fault detection rate. Therefore, in our study, we used a simple greedy algorithm to instantiate the CCCP prioritization function for statement, branch, and method coverage criteria. As we just want to evaluate the performance of code combinations coverage against traditional code coverage (e.g. statement) and the *additional* strategy has been widely accepted as the most effective prioritization strategy, we thus implemented greedy strategies based on the work of Rothermel et al. [1] as the control techniques for evaluation of our proposed approaches.

3.2. Code Combinations Coverage

Various RTCP approaches, based on different prioritization strategies, have been proposed to reduce regression testing overheads. Many of these approaches used individual code unit coverage of a test case to guide the prioritization process. For example, greedy strategies only take the number of covered code units into account, with the code units considered as parameters, or individually, and in isolation. However, this may lead to a loss of coverage information, and regression testing has traditionally used historical testing information to guide future testing. Thus, the degree to which the information is used is significant for regression testing, and if we consider the combination between code units, it may be possible to devise strategies to take further advantage of the code coverage information. In our hypothesis, the code units are related, not isolated, and faults may be triggered by combinations amongst them. Based on this, we can make use of more accurate testing information than traditional RTCP approaches to guide the prioritization process.

In our work, a code unit is a general term describing one structural code element — a statement, branch, or method. Consider a program P that has m code units (statements, branches, or methods) that form the code unit set $U = \{u_1, u_2, \cdots, u_m\}$, and a regression test set T with n test cases ($T = \{tc_1, tc_2, \cdots, tc_n\}$). We define a function $isCovered(tc, u)$ to measure whether or not a test case tc covers the code unit u, as follows:

$$isCovered(tc, u) = \begin{cases} 1, & \textit{if } u \textit{ is covered by } tc, \\ 0, & \textit{otherwise} \end{cases} \quad (3)$$

As a result, each test case tc can be represented by an m-wise binary array through the $convertTest$ function: $convertTest(tc) = (isCovered(tc, u_1), isCovered(tc, u_2), \cdots, isCovered(tc, u_m))$. For ease of description, with the increase of i for each u_i in U, we make use of the incremental values to describe whether or not tc covers the code unit u_i: $isCovered(tc, u_i)$, where $1 \le i \le m$, defined as:

$$isCovered(tc, u_i) = \begin{cases} 2i - 1, & \textit{if } u_i \textit{ is covered by } tc, \\ 2i, & \textit{otherwise} \end{cases} \quad (4)$$

In other words, an odd number represents the situation where the code unit in question is covered by a given test case; and an even number means that it is not covered. For example, using Equation (3), the values $(1, 1, 1, 0, 0)$ for a test case tc would mean that the first three code units are covered by tc, but that the last two are not. Equation (4) would allow this to be represented as $tc = (1, 3, 5, 8, 10)$. In effect, each code unit can be considered a parameter that contains binary parameter values (an odd number and an even number): the first code unit takes the value 1 or 2; the second code unit takes 3 or 4; and so on. In other words, each test case becomes an abstract test case (as defined in Section 2.1). The λ-wise *code combinations coverage* (CCC) value of tc against the test set T is defined as the number of λ-wise code-unit combinations covered by tc that are not covered by T:

$$CCC(tc, T, \lambda) = |CombSet(convertTest(tc), \lambda) \setminus CombSet(T', \lambda)| \tag{5}$$

where $T' = \bigcup_{tc \in T}\{convertTest(tc)\}$.

3.3. Code Combinations Coverage based Prioritization

In our model, we view CCCP as a general strategy that can be applied to different prioritization algorithms using different coverage criteria. As greedy strategies are among the most widely-adopted prioritization strategies [1, 10], and the *additional* greedy strategy is considered to be one of the most effective RTCP approaches [3, 17, 18, 20], in terms of fault detection rate, we adopted a simple greedy strategy to instantiate the function for the proposed code combinations coverage.

Generally speaking, the approach chooses an element from candidates as the next test case such that it covers the largest number of λ-wise code-unit-value combinations that have not already been covered by previously selected test cases. Accordingly, the test case with the maximum number of uncovered code-unit-value combinations compared with the already selected test cases

Algorithm 1 ***calculateCombinations***$(temp\_cover, \lambda)$

Input: *temp_cover*: List of code unit values, λ: Combination strength
Output: *Combinations*: Set of λ-wise code-unit-value combinations

```
Combinations ← ∅
if λ = 1 then
    for each item ∈ temp_cover do
        Combinations ← Combinations ⋃ {item}
    end for
    return Combinations
end if
num ← |temp_cover|                                  ▷ The number of code units
for each i (0 ≤ i ≤ num − λ) do
    for each item ∈ Calculate(temp_cover.sublist(i + 1), λ − 1) do
        Combinations ← Combinations ⋃ {temp_cover.get(i) + item}
    end for
end for
return Combinations
```

Algorithm 2 Pseudocode of CCCP

Input: T: $\{tc_1, tc_2, \cdots, tc_n\}$ is a set of unordered test cases with size n, λ: Combination strength, U: $\{u_1, u_2, \cdots, u_m\}$ is a set of m code units for the program P
Output: S: Prioritized test cases

```
1: maximum ← −1
2: k ← 1
3: for each i (1 ≤ i ≤ n) do
4:     Selected[i] ← false
5:     num ← 0
6:     for each j (1 ≤ j ≤ m) do
7:         Cover[i, j] ← isCovered(tc_i, u_j)
8:         temp_cover[j] ← Cover[i, j]
9:         if Cover[i, j]%2 == 1 then
10:            num ← num + 1
11:        end if
12:    end for
13:    if num > maximum then
14:        maximum ← num
15:        k ← i
16:    end if
17:    Combinations[i] ← calculateCombinations(temp_cover, λ)   ▷ Calculate λ-wise code-unit-value combinations covered by tc_i
18:    UncoverCombinations ← UncoverCombinations ⋃ Combinations[i]
19:    Selected[k] ← true
20: end for
21: tempCombinations ← UncoverCombinations
22: S ← S > ⟨tc_k⟩    ▷ Choose a candidate covering the maximum number of code units, and then append it to S
23: UncoverCombinations ← UncoverCombinations \ Combinations[k]
24: flag ← false
25: while |T| ≠ |S| do
26:     if flag then                                   ▷ Restart the process
27:         UncoverCombinations ← tempCombinations
28:     end if
29:     maximum ← −1
30:     k ← 1
31:     for each i (1 ≤ i ≤ n) do
32:         if not Selected[i] then
33:             num ← |UncoverCombinations ⋂ Combinations[i]|
34:             if num > maximum then
35:                 maximum ← num
36:                 k ← i
37:             end if
38:         end if
39:     end for
40:     if maximum == 0 then    ▷ No uncovered code-unit-value combinations
41:         flag ← true
42:     else
43:         flag ← false
44:     end if
45:     S ← S > ⟨tc_k⟩    ▷ Choose the best candidate, and then append it to S
46:     UncoverCombinations ← UncoverCombinations \ Combinations[k]
47:     Selected[k] ← true
48: end while
49: return S
```

T will be selected: $CCC(tc, T, \lambda)$, from Equation (5). Algorithm 1 provides the detailed *CombSet* calculation process. Furthermore, when there are no further new code-unit-value combinations, then all remaining test cases are prioritized according the previous process, in a manner similar to the *additional* greedy strategy.

Algorithm 2 formally presents the pseudocode of C-CCP. A Boolean array $Selected[i]$ $(1 \le i \le n)$ denotes whether or not test case tc_i has been selected for prioritization; and another Boolean array $Cover[i, j]$ $(1 \le i \le n, 1 \le j \le m)$ identifies whether or not test case tc_i covers the code unit u_j. Similarly, an array $Combinations[i]$

stores the λ-wise code-unit-value combinations covered by tc_i; and a set *UncoverCombinations* stores all uncovered λ-wise code-unit-value combinations. In addition, the variable $flag$ indicates whether or not all λ-wise code-unit-value combinations have been covered.

In Algorithm 2, Lines 1–24 perform initialization, and also choose the first test case from the candidates; while Lines 25–49 prioritize the test cases. More specifically, since each candidate test case covers the same number of uncovered λ-wise code-unit-value combinations before prioritization, our approach follows the total and additional test prioritization techniques to choose the first test case: the one covering the largest number of units (Lines 9–16). Then, the number of uncovered λ-wise code-unit-value combinations against previously selected test cases (S) is calculated for each remaining test case, and a candidate with the maximum value is selected as the next test case is appended to S (Lines 31–39). Before choosing the next test case, our approach examines whether or not there are any λ-wise code-unit-value combinations that are not covered by the test cases in S: If there are not, the remaining candidate test cases are prioritized by restarting the previous process (Lines 26–28). Once an element is selected as the next test case, our approach updates the set of uncovered λ-wise code-unit-value combinations (Lines 23 and 46). This process is repeated until all elements from T have been added to S. Similar to *additional* test prioritization, when facing a tie where more than one test case has the largest number of uncovered code-unit-value combinations, our approach randomly selects one.

To further explain the details of the proposed approach, Figure 1 illustrates an example of the CCCP process with $\lambda = 1$. Similar to the total and additional test prioritization techniques, CCCP chooses the first test case that covers the largest number of code units (the maximum amount of odd numbers). Since there are two candidates with the maximum number of code units, tc_1 and tc_2 (both covering three code units), CCCP randomly chooses one of them (in this case, tc_1), and adds it to S. CCCP then updates the set *UncoverCombinations*, and calculates the CCC value for each candidate: $CCC(tc_2, S, 1) = 2$ and $CCC(tc_3, S, 1) = 3$. Since tc_3 has the greater CCC value, it is selected as the next test case, and added to S. In contrast, the *total* prioritization technique would choose tc_2 as the second test case, because tc_2 covers more code units than tc_3; and the *additional* technique would randomly select one from tc_2 and tc_3 as the second test case, because both candidates cover the same number of uncovered code units. Finally, in our approach, the last candidate tc_2 is added to S, resulting in $S = \langle tc_1, tc_3, tc_2 \rangle$.

Suppose that an unordered test set $T = \{tc_1, tc_2, tc_3\}$, the unit set $\{u_1, u_2, u_3, u_4\}$, and the following matrix based on the function $isCovered(tc_i, u_j)$, where $1 \le i \le 3$ and $1 \le j \le 4$.

	u_1	u_2	u_3	u_4
tc_1	1	4	5	7
tc_2	2	3	5	7
tc_3	1	3	6	8

Consider CCCP with $\lambda = 1$, $UncoverCombinations = \{(1), (2), (3), (4), (5), (6), (7), (8)\}$.
Step 1: Randomly choose an element from $\{tc_1, tc_2\}$ as the first test case in S, because each test case covers the same number of units, for example, $S = < tc_1 >$.
Step 2: Update $UncoverCombinations = \{(2), (3), (6), (8)\}$, and then calculate
$CombSet(tc_2, 1) = \{(2), (3), (5), (7)\}$;
$CombSet(tc_3, 1) = \{(1), (3), (6), (8)\}$;
$CCC(tc_2, S, 1) = |UncoverCombinations \cap CombSet(tc_2, 1)| = |\{(2), (3)\}| = 2$;
$CCC(tc_3, S, 1) = |UncoverCombinations \cap CombSet(tc_3, 1)| = |\{(3), (6), (8)\}| = 3$.
As a result, tc_3 is selected as the next test case for S, i.e., $S = < tc_1, tc_3 >$.
Step 3: Add the last test case tc_2 for S, i.e., $S = < tc_1, tc_3, tc_2 >$.

Fig. 1: An illustrative example of CCCP

4. Empirical Study

In this section, we present our empirical study, including the research questions underlying the study. We also discuss some independent and dependent variables, and explain the subject programs, test suites, and experimental setup in detail.

4.1. Research Questions

Due to space limitations and practical performance constraints (higher λ values may result in more substantial running time), we present the evaluation of our proposed approach's performance when $\lambda = 1$ and 2. Unless explicitly stated, $\lambda = 1$ is used as the default value for CCCP. The empirical study was conducted to answer the following six research questions.

RQ1 How does CCCP compare with other RTCP approaches in terms of testing effectiveness measured by the fault detection rates?

RQ2 How does CCCP compare with other RTCP approaches in terms of testing effectiveness measured by the cost-cognizant fault detection rates?

RQ3 How does the granularity of code coverage impact the comparative effectiveness of CCCP?

RQ4 How does the granularity of test cases impact on the comparative effectiveness of CCCP?

RQ5 How does the efficiency of CCCP compare with other RTCP approaches, in terms of execution time?

RQ6 How does the use of code combinations coverage with $\lambda = 2$ impact on the testing effectiveness of CCCP?

4.2. Independent Variables

In this study, we consider the following three independent variables.

4.2.1. Prioritization Techniques

Since our proposed CCCP technique takes advantage of the information about dynamic coverage and test cases as inputs, for a fair comparison, it is necessary to choose other RTCP techniques that use similar information to guide the test cases prioritization. In this study, we selected four such RTCP techniques: *total test prioritization* [1], *additional test prioritization* [1], *adaptive random test prioritization* [3], and *search-based test prioritization* [2]. The total test prioritization technique prioritizes test cases based on the descending number of code units covered by those tests. The additional technique, in contrast, greedily chooses each element from candidates such that it covers the largest number of code units not yet covered by the previously selected tests. The adaptive random technique greedily selects each element from random candidates such that it has the greatest maximum distance from the already selected tests. Finally, the search-based technique considers all permutations as candidate solutions, and uses a meta-heuristic search algorithm to guide the search for a better test execution order — a genetic algorithm was adopted in this study, due to its effectiveness [2]. These four test prioritization techniques, whose details are presented in Table 1, have been widely used in RTCP previous studies [17, 18, 20].

4.2.2. Code Coverage Granularity

When using code coverage information to support RTCP, the coverage granularity can be considered a constituent part of the prioritization techniques. To enable sufficient evaluations, we used structural coverage criteria at the statement-, branch-, and method-coverage levels.

Table 1: Studied RTCP techniques

Mnemonic	Description	Reference
TCP_{tot}	Greedy total test prioritization	[1]
TCP_{add}	Greedy additional test prioritization	[1]
TCP_{art}	Adaptive random test prioritization	[3]
TCP_{search}	Search-based test prioritization	[2]
TCP_{ccc}	Our proposed CCCP technique	This study

4.2.3. Test Case Granularity

For the subject programs written in Java, we considered an additional factor in the prioritization techniques, the test-case granularity. Test-case granularity is at either the test-class level, or the test-method level. For the test-class level, each JUnit test-case class was a test case; for the test-method level, each test method in a JUnit test-case class was a test case. In other words, a test case at the test-class level generally involves a number of test cases at the test-method level. For C subject programs, however, the actual program inputs were the test cases.

4.3. Dependent Variables

Because we were examining the fault detection capability, we adopted the widely-used APFD (*average percentage faults detected*) as the evaluation metric for fault detection rates [1]. Given a test suite T, with n test cases. If T' is a permutation of T, in which TF_i is the position of first test case that reveals the fault i, then the APFD value for T' is given by the following equation:

$$APFD = 1 - \frac{\sum_{i=1}^{m} TF_i}{n * m} + \frac{1}{2n} \tag{6}$$

Although APFD has often been used to evaluate RTCP techniques, it assumes that each test case incurs the same time cost, an assumption that may not hold up in practice. Elbaum et al. [21], therefore, introduced an APFD variant, APFDc, a cost-cognizant version of APFD that considers both the fault severity and the test case execution cost. Similar to APFD, APFDc has also been applied to the evaluation of RTCP techniques, resulting in a more comprehensive evaluation [22]. APFDc is defined as:

$$APFD_c = \frac{\sum_{i=1}^{m}(\alpha_i \times (\sum_{j=TF_i}^{n} \beta_j - \frac{1}{2}\alpha_i))}{\sum_{i=1}^{m} \alpha_i \times \sum_{i=1}^{n} \beta_i} \tag{7}$$

where $\alpha_1, \alpha_2, \cdots, \alpha_m$ are the severities of the m detected faults, $\beta_1, \beta_2, \cdots, \beta_n$ are the execution costs of n test cases, and TF_i has the same meaning as in APFD. Because of the difficulty involved in estimating fault severity, following previous studies [22], we assumed that all faults had the same severity level for this study. Accordingly, the definition of APFDc can be described as:

$$APFD_c = \frac{\sum_{i=1}^{m}(\sum_{j=TF_i}^{n} \beta_j - \frac{1}{2}\beta_{TF_i})}{m \times \sum_{j=1}^{n} \beta_j} \tag{8}$$

Intuitively speaking, prioritized test suites that both find faults faster and require less execution time, will have higher APFDc values.

Table 2: Subject program details

Language	Program	Version	KLoC	#Branch	#Method	#Class	#Test_Case		#Mutant		#Subsuming_Mutant	
							#T_Class	#T_Method	#All	#Detected	#SM_Class	#SM_Method
Java	*ant_v1*	v1_9	25.80	5,240	2,511	228	34 (34)	137 (135)	6,498	1,332	59	32
	ant_v2	1.4	39.70	8,797	3,836	342	52 (52)	219 (214)	11,027	2,677	90	47
	ant_v3	1.4.1	39.80	8,831	3,845	342	52 (52)	219 (213)	11,142	2,661	92	47
	jmeter_v1	v1_7_3	33.70	3,815	2,919	334	26 (21)	78 (61)	8,850	573	38	20
	jmeter_v2	v1_8	33.10	3,799	2,838	319	29 (24)	80 (74)	8,777	867	37	22
	jmeter_v3	v1_8_1	37.30	4,351	3,445	373	33 (27)	78 (77)	9,730	1,667	47	25
	jmeter_v4	v1_9_RC1	38.40	4,484	3,536	380	33 (27)	78 (77)	10,187	1,703	47	25
	jmeter_v5	v1_9_RC2	41.10	4,888	3,613	389	37 (30)	97 (83)	10,459	1,651	53	29
	jtopas_v1	0.4	1.89	519	284	19	10 (10)	126 (126)	704	399	29	9
	jtopas_v2	0.5.1	2.03	583	302	21	11 (11)	128 (128)	774	446	34	10
	jtopas_v3	0.6	5.36	1,491	748	50	18 (16)	209 (207)	1,906	1,024	57	16
	xmlsec_v1	v1_0_4	18.30	3,534	1,627	179	15 (15)	92 (91)	5,501	1,198	32	12
	xmlsec_v2	v1_0_5D2	19.00	3,789	1,629	180	15 (15)	94 (94)	5,725	1,204	33	12
	xmlsec_v3	v1_0_71	16.90	3,156	1,398	145	13 (13)	84 (84)	3,833	1,070	27	10
C	*flex_v0*	2.4.3	8.96	2,005	138	–	500		–	–	–	
	flex_v1	2.4.7	9.47	2,011	147	–	500		13,873	6,177	32	
	flex_v2	2.5.1	12.23	2,656	162	–	500		14,822	6,396	32	
	flex_v3	2.5.2	12.25	2,666	162	–	500		775	420	20	
	flex_v4	2.5.3	12.38	2,678	162	–	500		14,906	6,417	33	
	flex_v5	2.5.4	12.37	2,680	162	–	500		14,922	6,418	32	
	grep_v0	2.0	8.16	3,420	119	–	144		–	–	–	
	grep_v1	2.2	11.99	3,511	104	–	144		23,896	3,229	56	
	grep_v2	2.3	12.72	3,631	109	–	144		24,518	3,319	58	
	grep_v3	2.4	12.83	3,709	113	–	144		17,656	3,156	54	
	grep_v4	2.5	20.84	2,531	102	–	144		17,738	3,445	58	
	grep_v5	2.7	58.34	2,980	109	–	144		17,108	3492	59	
	gzip_v0	1.0.7	4.32	1,468	81	–	156		–	–	–	
	gzip_v1	1.1.2	4.52	1,490	81	–	156		7,429	639	8	
	gzip_v2	1.2.2	5.05	1,752	98	–	156		7,599	659	8	
	gzip_v3	1.2.3	5.06	1,610	93	–	156		7,678	547	7	
	gzip_v4	1.2.4	5.18	1,663	93	–	156		7,838	548	7	
	gzip_v5	1.3	5.68	1,733	97	–	156		8,809	210	7	
	make_v0	3.75	17.46	4,397	181	–	111		–	–	–	
	make_v1	3.76.1	18.57	4,585	181	–	111		36,262	5,800	37	
	make_v2	3.77	19.66	4,784	190	–	111		38,183	5,965	29	
	make_v3	3.78.1	20.46	4,845	216	–	111		42,281	6,244	28	
	make_v4	3.79	23.13	5,413	239	–	111		48,546	6,958	29	
	make_v5	3.80	23.40	5,032	268	–	111		47,310	7,049	28	
	sed_v0	3.01	7.79	676	66	–	324		–	–	–	
	sed_v1	3.02	7.79	712	65	–	324		2,506	1,009	16	
	sed_v2	4.0.6	18.55	1,011	65	–	324		5,947	1,048	18	
	sed_v3	4.0.8	18.69	1,017	66	–	324		5,970	450	18	
	sed_v4	4.1.1	21.74	1,141	70	–	324		6,578	470	19	
	sed_v5	4.2	26.47	1,412	98	–	324		7,761	628	22	

4.4. Subject Programs, Test Suites and Faults

We conducted our study on 14 versions of four Java programs (three versions of *ant*; five versions of *jmeter*; three versions of *jtopas*; and three versions of *xmlsec*) downloaded from the *Software-artifact Infrastructure Repository* (SIR) [23, 24], and 30 versions of five real-life Unix utility programs, from the *GNU FTP server* [25]. Both the Java and C programs have been widely used as benchmarks to evaluate RTCP problems [3, 10, 19, 26]. Table 2 summarizes the subject program details, with Columns 3 to 7 presenting the version, size, number of branches, number of methods, and number of classes (including interfaces), respectively.

Each version of the Java programs has a JUnit test suite that was developed during the program's evolution. These test suites have two levels of test-case granularity: the test-class and the test-method. The numbers of JUnit test cases (including both test-class and test-method levels) are shown in the **#Test_Case** column, as **#T_ Class** and **#T_Method**: The data is presented as x (y), where x is the total number of test cases; and y is the number of test cases that can be successfully executed. The test suites for the C programs are available from the SIR [23, 24]: The number of tests cases in each suite is also shown in the **#Test_Case** column of Table 2.

Because the seeded-in SIR faults were easily detected and small in size, for both C and Java programs, we used mutation faults to evaluate the performance of the different techniques. Mutation faults have previously been identified as suitable for simulating real program faults [27, 28, 29], and have been widely applied to regression test prioritization evaluations [1, 10, 17, 18, 19, 20, 30]. Eleven mutation operators from the "NEW_DEFAULTS" group of the PIT mutation tool [31] were used to generate mutants for the Java

programs. These operators, whose detailed descriptions can be found on the PIT website [32], were: *conditionals boundary*, *increments*, *invert negatives*, *math*, *negate conditionals*, *void method calls*, *empty returns*, *false returns*, *null returns*, *primitive returns*, and *true returns*. We downloaded the mutants from previous RTCP studies [19] for the C programs, which had been generated using seven mutation operators from Andrews et al. [33]: *statement deletion*, *unary insertion*, *constant replacement*, *arithmetic operator replacement*, *logical operator replacement*, *bitwise logical operator replacement*, and *relational operator replacement*.

Equivalent mutants [34, 35] are functionally equivalent versions of the original program, and thus cannot be killed: no test case applied to both the mutant and the original program could result in different behavior. In our study, therefore, all equivalent mutants were removed, leaving only those mutants that could be detected by at least one test case. In Table 2, the **#Mutant** column gives the total number of all mutants (**#All**), and the (**#Detected**) column gives the number of detected mutants. Although all detected mutants were considered in our study, some mutants, called *duplicated mutants*) [35], were equivalent to other mutants (but not to the original program). Similarly, some mutants, called *subsumed mutants* [36, 37] were subsumed by others: If a *subsuming mutant* [38]) is killed, then its subsumed mutants are also killed. We used the *Subsuming Mutants Identification* (SMI) algorithm [38] to remmove the duplicate and subsuming mutants in our fault set. SMI first removed duplicate mutants, and then greedily identified the most subsuming mutants — those mutants which, when killed, result in the highest number of other mutants being "collaterally" killed. The **#Subsuming_Mutant** column gives the number of subsuming mutants used in our study (the subsuming faults are classified as either test-class level (**#SM_Class**) or test-method level (**#SM_Method**) for the Java programs).

4.5. Experimental Setup

The experiments were conducted on a Linux 4.4.0-170-generic server with two Intel(R) Xeon(R) Platinum 8163 CPUs (2.50 GHz, two cores) and 16 GBs of RAM.

The Java programs were compiled using Java 1.8 [39]. The coverage information for each program version was obtained using the `FaultTracer` tool [40, 41], which, based on the ASM bytecode manipulation and analysis framework [42], uses on-the-fly bytecode instrumentation without any modification of the target program.

There were six versions of each C program: P_{V0}, P_{V1}, P_{V2}, P_{V3}, P_{V4}, and P_{V5}. Version P_{V0} was compiled using `gcc` 5.4.0 [43], and then the coverage information was obtained using the `gcov` tool [44], which is one of the `gcc` utilities.

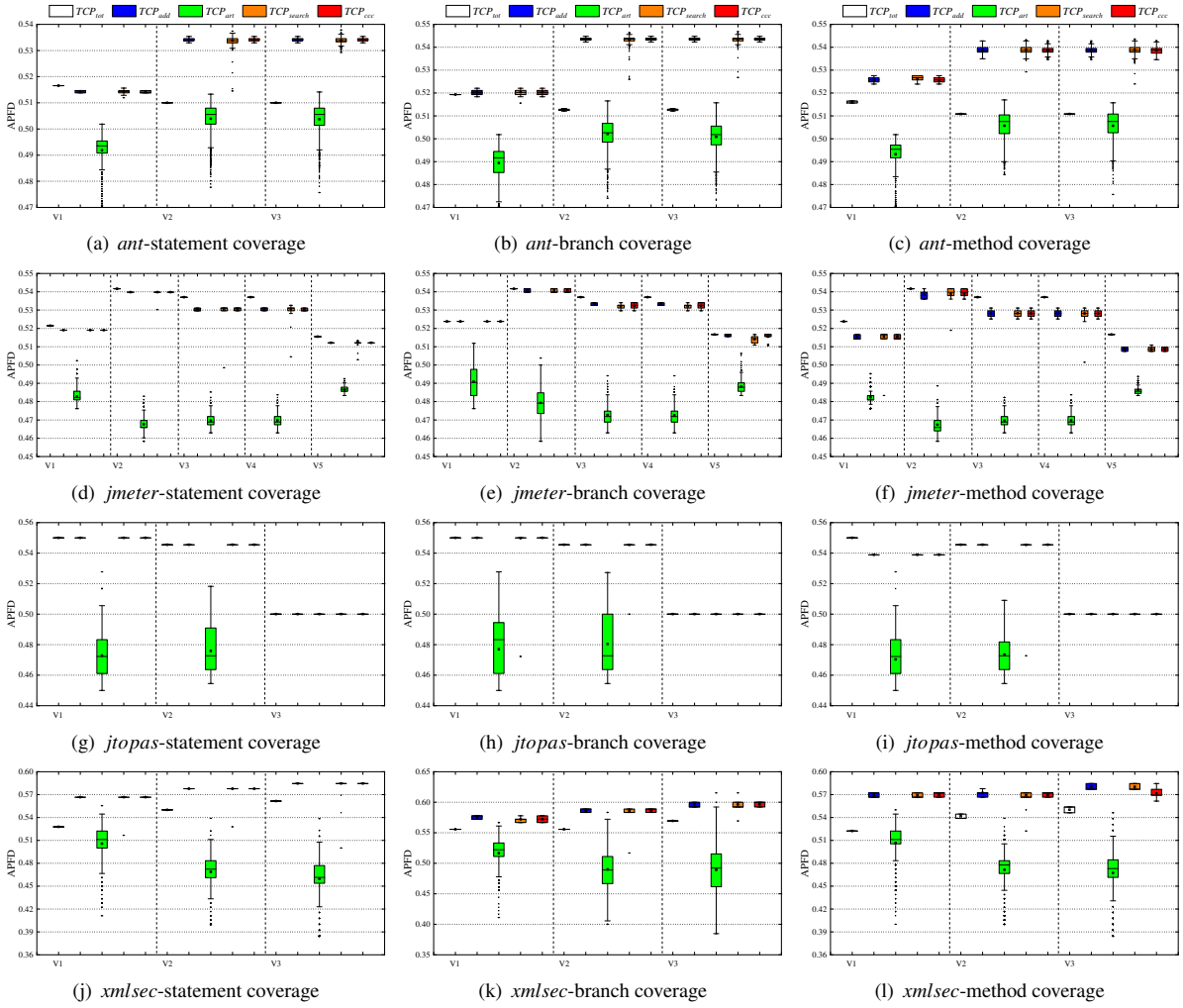
After collecting the code coverage information, we implemented all RTCP techniques in Java, and applied them to each program version under study, for all coverage criteria. Because the approaches contain randomness, each execution was repeated 1000 times. This resulted in, for each testing scenario, 1000 APFD or $APFD_c$ values (1000 orderings) for each RTCP approach. To test whether there was a statistically significant difference between CCCP and the other RTCP approaches, we performed the unpaired two-tailed Wilcoxon-Mann-Whitney test, at a significance level of 5%, following previously reported guidelines for inferential statistical analyses involving randomized algorithms [45, 46]. To identify which approach was better, we also calculated the effect size, measured by the non-parametric Vargha and Delaney effect size measure [47], $\hat{A}_{12}$, where $\hat{A}_{12}(X, Y)$ gives probability that the technique X is better than technique Y. The statistical analyses were performed using `R` [48].

5. Results and Analysis

This section presents the experimental results to answer the research questions.

To answer RQ1 to RQ4, Figures 2 to 8 present box plots of the distribution of the APFD or $APFD_c$ values (averaged over 1000 iterations). Each box plot shows the mean (square in the box), median (line in the box), and upper and lower quartiles (25th and 75th percentile) for the APFD or $APFD_c$ values for the RTCP techniques. Statistical analyses are also provided in Tables 3 to 11 for each pairwise APFD or $APFD_c$ comparison between CCCP and the other RTCP techniques. For example, for a comparison between two methods TCP_{ccc} and M, where $M \in \{TCP_{tot}, TCP_{add}, TCP_{art}, TCP_{search}\}$, the symbol ✔ means that TCP_{ccc} is better (p-value is less than 0.05, and the effect size $\hat{A}_{12}(TCP_{ccc}, M)$ is greater than 0.50); the symbol ✖ means that M is better (the p-value is less than 0.05, and $\hat{A}_{12}(TCP_{ccc}, M)$ is less than 0.50); and the symbol ❍ means that there is no statistically significant difference between them (the p-value is greater than 0.05).

To answer RQ5, Table 12 provides comparisons of the execution times for the different RTCP techniques. To answer RQ6, Figure 9 shows the APFD and $APFD_c$ results for CCCP with $\lambda = 2$, and Table 13 presents the corresponding statistical analysis.



(a) *ant*-statement coverage (b) *ant*-branch coverage (c) *ant*-method coverage

(d) *jmeter*-statement coverage (e) *jmeter*-branch coverage (f) *jmeter*-method coverage

(g) *jtopas*-statement coverage (h) *jtopas*-branch coverage (i) *jtopas*-method coverage

(j) *xmlsec*-statement coverage (k) *xmlsec*-branch coverage (l) *xmlsec*-method coverage

Fig. 2: **Effectiveness: APFD** results for **Java programs** at the **test-class level**

*5.1. **RQ1: Effectiveness of CCCP Measured by APFD***

Here, we provide the APFD results for CCCP for different code coverage and test case granularities. Figures 2 to 4 show the APFD results for the Java programs at the test-class level; at the test-method level; and the C programs, respectively. Each sub-figure in these figures has the program versions across the *x*-axis, and the APFD values for the five RTCP techniques on the *y*-axis. Tables 3 to 5 present the corresponding statistical comparisons.

5.1.1. APFD Results: Java Programs (Test-Class Level)

Based on Figure 2 and Table 3, we have the following observations:

(1) Compared with the total test prioritization technique, CCCP achieves better performances for the program *xmlsec*, irrespective of code coverage granularity, with differences between the mean and median APFD values reaching about 3%. For the other programs (*ant*, *jmeter*, and *jtopas*), however, they have very similar APFD results.

(2) CCCP performs much better than adaptive random test prioritization, regardless of subject program and code coverage granularity, with the maximum mean and median APFD differences reaching about 12%.

(3) CCCP has very similar performance to the additional and search-based test prioritization techniques, with the mean and median APFD differences approximately equal to 1%.

Table 3: Statistical **effectiveness** comparisons of **APFD** for **Java programs** at the **test-class level**. For a comparison between two methods TCP_{ccc} and M, where $M \in \{TCP_{tot}, TCP_{add}, TCP_{art}, TCP_{search}\}$, the symbol ✔ means that TCP_{ccc} is better (p-value is less than 0.05, and the effect size $\hat{A}_{12}(TCP_{ccc}, M)$ is greater than 0.50); the symbol ✖ means that M is better (the p-value is less than 0.05, and $\hat{A}_{12}(TCP_{ccc}, M)$ is less than 0.50); and the symbol ❍ means that there is no statistically significant difference between them (the p-value is greater than 0.05).

Program Name	Statement Coverage				Branch Coverage				Method Coverage			
	TCP_{tot}	TCP_{add}	TCP_{art}	TCP_{search}	TCP_{tot}	TCP_{add}	TCP_{art}	TCP_{search}	TCP_{tot}	TCP_{add}	TCP_{art}	TCP_{search}
ant_v1	✖(0.00)	❍(0.49)	✔(1.00)	❍(0.50)	✔(0.76)	❍(0.51)	✔(1.00)	❍(0.49)	✔(1.00)	❍(0.49)	✔(1.00)	✖(0.36)
ant_v2	✔(1.00)	❍(0.51)	✔(1.00)	✔(0.57)	✔(1.00)	❍(0.49)	✔(1.00)	❍(0.52)	✔(1.00)	❍(0.49)	✔(1.00)	✖(0.47)
ant_v3	✔(1.00)	❍(0.50)	✔(1.00)	✔(0.59)	✔(1.00)	❍(0.50)	✔(1.00)	✔(0.54)	✔(1.00)	❍(0.49)	✔(1.00)	✖(0.46)
ant	✔(0.89)	❍(0.50)	✔(1.00)	✔(0.53)	✔(0.97)	❍(0.50)	✔(1.00)	❍(0.51)	✔(1.00)	❍(0.49)	✔(1.00)	✖(0.47)
jmeter_v1	✖(0.00)	❍(0.50)	✔(1.00)	❍(0.50)	❍(0.50)	❍(0.50)	✔(1.00)	❍(0.50)	✖(0.00)	❍(0.50)	✔(1.00)	✖(0.46)
jmeter_v2	✖(0.00)	❍(0.50)	✔(1.00)	❍(0.50)	✖(0.24)	❍(0.51)	✔(1.00)	❍(0.50)	✖(0.14)	✔(0.53)	✔(1.00)	❍(0.49)
jmeter_v3	✖(0.00)	❍(0.51)	✔(1.00)	❍(0.48)	✖(0.00)	✖(0.34)	✔(1.00)	✔(0.59)	✖(0.00)	❍(0.50)	✔(1.00)	✖(0.47)
jmeter_v4	✖(0.00)	✖(0.47)	✔(1.00)	✖(0.46)	✖(0.00)	✖(0.33)	✔(1.00)	✔(0.57)	✖(0.00)	❍(0.49)	✔(1.00)	❍(0.48)
jmeter_v5	✖(0.00)	❍(0.50)	✔(1.00)	❍(0.50)	✖(0.32)	✖(0.43)	✔(1.00)	✔(0.72)	✖(0.00)	❍(0.51)	✔(1.00)	❍(0.50)
jmeter	✖(0.36)	❍(0.50)	✔(1.00)	❍(0.50)	✖(0.40)	✖(0.47)	✔(1.00)	✔(0.52)	✖(0.32)	❍(0.50)	✔(1.00)	❍(0.49)
jtopas_v1	❍(0.50)	❍(0.50)	✔(1.00)	❍(0.50)	❍(0.50)	❍(0.50)	✔(1.00)	❍(0.50)	✖(0.00)	❍(0.50)	✔(1.00)	❍(0.50)
jtopas_v2	❍(0.50)	❍(0.50)	✔(1.00)	❍(0.50)	❍(0.50)	❍(0.50)	✔(1.00)	❍(0.50)	❍(0.50)	❍(0.50)	✔(1.00)	❍(0.50)
jtopas_v3	❍(0.50)	❍(0.50)	❍(0.50)	❍(0.50)	❍(0.50)	❍(0.50)	❍(0.50)	❍(0.50)	❍(0.50)	❍(0.50)	❍(0.50)	❍(0.50)
jtopas	❍(0.50)	❍(0.50)	✔(0.93)	❍(0.50)	❍(0.50)	❍(0.50)	✔(0.91)	❍(0.50)	✖(0.33)	❍(0.50)	✔(0.94)	❍(0.50)
xmlsec_v1	✔(1.00)	❍(0.50)	✔(1.00)	❍(0.50)	✔(1.00)	✖(0.31)	✔(1.00)	❍(0.52)	✔(1.00)	❍(0.51)	✔(1.00)	✔(0.54)
xmlsec_v2	✔(1.00)	❍(0.50)	✔(1.00)	❍(0.50)	✔(1.00)	❍(0.50)	✔(1.00)	❍(0.51)	✔(1.00)	✖(0.32)	✔(1.00)	✔(0.54)
xmlsec_v3	✔(1.00)	❍(0.50)	✔(1.00)	❍(0.50)	✔(1.00)	❍(0.52)	✔(1.00)	❍(0.51)	✔(1.00)	✖(0.16)	✔(1.00)	✖(0.18)
xmlsec	✔(1.00)	❍(0.50)	✔(1.00)	❍(0.50)	✔(0.97)	✖(0.48)	✔(1.00)	❍(0.50)	✔(1.00)	✖(0.31)	✔(1.00)	✖(0.40)
All Java Programs	✔(0.58)	❍(0.50)	✔(0.98)	❍(0.50)	✔(0.58)	❍(0.50)	✔(0.96)	❍(0.50)	✔(0.59)	✖(0.49)	✔(0.97)	✖(0.49)

Table 4: Statistical **effectiveness** comparisons of **APFD** for **Java programs** at the **test-method level**. For a comparison between two methods TCP_{ccc} and M, where $M \in \{TCP_{tot}, TCP_{add}, TCP_{art}, TCP_{search}\}$, the symbol ✔ means that TCP_{ccc} is better (p-value is less than 0.05, and the effect size $\hat{A}_{12}(TCP_{ccc}, M)$ is greater than 0.50); the symbol ✖ means that M is better (the p-value is less than 0.05, and $\hat{A}_{12}(TCP_{ccc}, M)$ is less than 0.50); and the symbol ❍ means that there is no statistically significant difference between them (the p-value is greater than 0.05).

Program Name	Statement Coverage				Branch Coverage				Method Coverage			
	TCP_{tot}	TCP_{add}	TCP_{art}	TCP_{search}	TCP_{tot}	TCP_{add}	TCP_{art}	TCP_{search}	TCP_{tot}	TCP_{add}	TCP_{art}	TCP_{search}
ant_v1	✔(1.00)	✔(0.53)	✔(1.00)	✔(0.88)	✔(1.00)	❍(0.48)	✔(1.00)	✔(0.91)	✔(1.00)	❍(0.50)	✔(0.98)	✔(0.72)
ant_v2	✔(1.00)	❍(0.51)	✔(1.00)	✔(1.00)	✔(1.00)	✖(0.46)	✔(1.00)	✔(1.00)	✔(1.00)	❍(0.50)	✔(1.00)	✔(0.90)
ant_v3	✔(1.00)	❍(0.50)	✔(1.00)	✔(1.00)	✔(1.00)	❍(0.51)	✔(1.00)	✔(1.00)	✔(1.00)	❍(0.51)	✔(1.00)	✔(0.90)
ant	✔(1.00)	❍(0.51)	✔(1.00)	✔(0.98)	✔(1.00)	❍(0.49)	✔(1.00)	✔(0.97)	✔(1.00)	❍(0.50)	✔(1.00)	✔(0.84)
jmeter_v1	✔(1.00)	❍(0.50)	✔(1.00)	✔(0.62)	✔(1.00)	✖(0.30)	✔(1.00)	✔(0.84)	✔(0.56)	❍(0.50)	✔(0.98)	✖(0.47)
jmeter_v2	✔(1.00)	✔(0.96)	✔(1.00)	✔(0.68)	✔(1.00)	❍(0.49)	✔(1.00)	✔(0.82)	✔(0.90)	❍(0.51)	✔(0.98)	❍(0.48)
jmeter_v3	✔(1.00)	✔(0.55)	✔(1.00)	✔(0.82)	✔(1.00)	❍(0.48)	✔(1.00)	✔(0.64)	✔(1.00)	✖(0.26)	✔(1.00)	✔(0.55)
jmeter_v4	✔(1.00)	✔(0.60)	✔(1.00)	✔(0.84)	✔(1.00)	❍(0.49)	✔(1.00)	✔(0.64)	✔(1.00)	✖(0.25)	✔(1.00)	✔(0.56)
jmeter_v5	✔(1.00)	❍(0.51)	✔(1.00)	✔(0.89)	✔(1.00)	✖(0.45)	✔(1.00)	✔(0.76)	✔(1.00)	✖(0.32)	✔(1.00)	✔(0.64)
jmeter	✔(1.00)	✔(0.54)	✔(1.00)	✔(0.63)	✔(1.00)	✖(0.48)	✔(1.00)	✔(0.68)	✔(0.86)	✖(0.43)	✔(0.98)	✔(0.52)
jtopas_v1	✔(1.00)	✔(0.98)	✖(0.00)	✔(0.77)	✔(1.00)	✖(0.46)	✖(0.00)	✖(0.43)	✔(1.00)	❍(0.49)	✔(0.73)	❍(0.50)
jtopas_v2	✔(1.00)	✔(0.79)	✖(0.00)	✔(0.53)	✔(1.00)	✔(0.89)	✖(0.00)	✔(0.70)	✔(1.00)	❍(0.50)	✖(0.27)	✖(0.35)
jtopas_v3	✔(1.00)	❍(0.50)	✖(0.00)	✖(0.16)	✔(1.00)	❍(0.49)	✖(0.00)	✖(0.31)	✔(1.00)	❍(0.49)	✔(0.67)	✔(0.62)
jtopas	✔(1.00)	✔(0.59)	✖(0.00)	❍(0.51)	✔(1.00)	✔(0.54)	✖(0.00)	✖(0.47)	✔(1.00)	❍(0.50)	✔(0.55)	✖(0.48)
xmlsec_v1	✔(1.00)	❍(0.49)	✔(1.00)	✖(0.37)	✔(1.00)	❍(0.49)	✔(1.00)	✔(0.78)	✔(1.00)	✔(0.53)	✔(1.00)	✔(0.58)
xmlsec_v2	✔(1.00)	❍(0.49)	✔(1.00)	❍(0.50)	✔(1.00)	❍(0.50)	✔(1.00)	✔(0.77)	✔(1.00)	❍(0.51)	✔(1.00)	✔(0.60)
xmlsec_v3	✔(1.00)	✖(0.46)	✔(1.00)	❍(0.52)	✔(1.00)	✖(0.30)	✔(1.00)	✔(0.63)	✔(1.00)	❍(0.51)	✔(1.00)	✔(0.56)
xmlsec	✔(1.00)	❍(0.49)	✔(1.00)	✖(0.47)	✔(1.00)	✖(0.43)	✔(1.00)	✔(0.72)	✔(1.00)	❍(0.51)	✔(1.00)	✔(0.56)
All Java Programs	✔(0.97)	✔(0.51)	✔(0.70)	✔(0.57)	✔(0.90)	✖(0.49)	✔(0.59)	✔(0.59)	✔(0.96)	✖(0.49)	✔(0.76)	✔(0.54)

(4) There is a statistically significant difference between TCP_{ccc} and TCP_{art}, which supports the above observations. However, none of the other three techniques (TCP_{tot}, TCP_{add}, or TCP_{search}) is either always better or always worse than TCP_{ccc}, with TCP_{ccc} sometimes performing better for some programs, and sometimes worse.

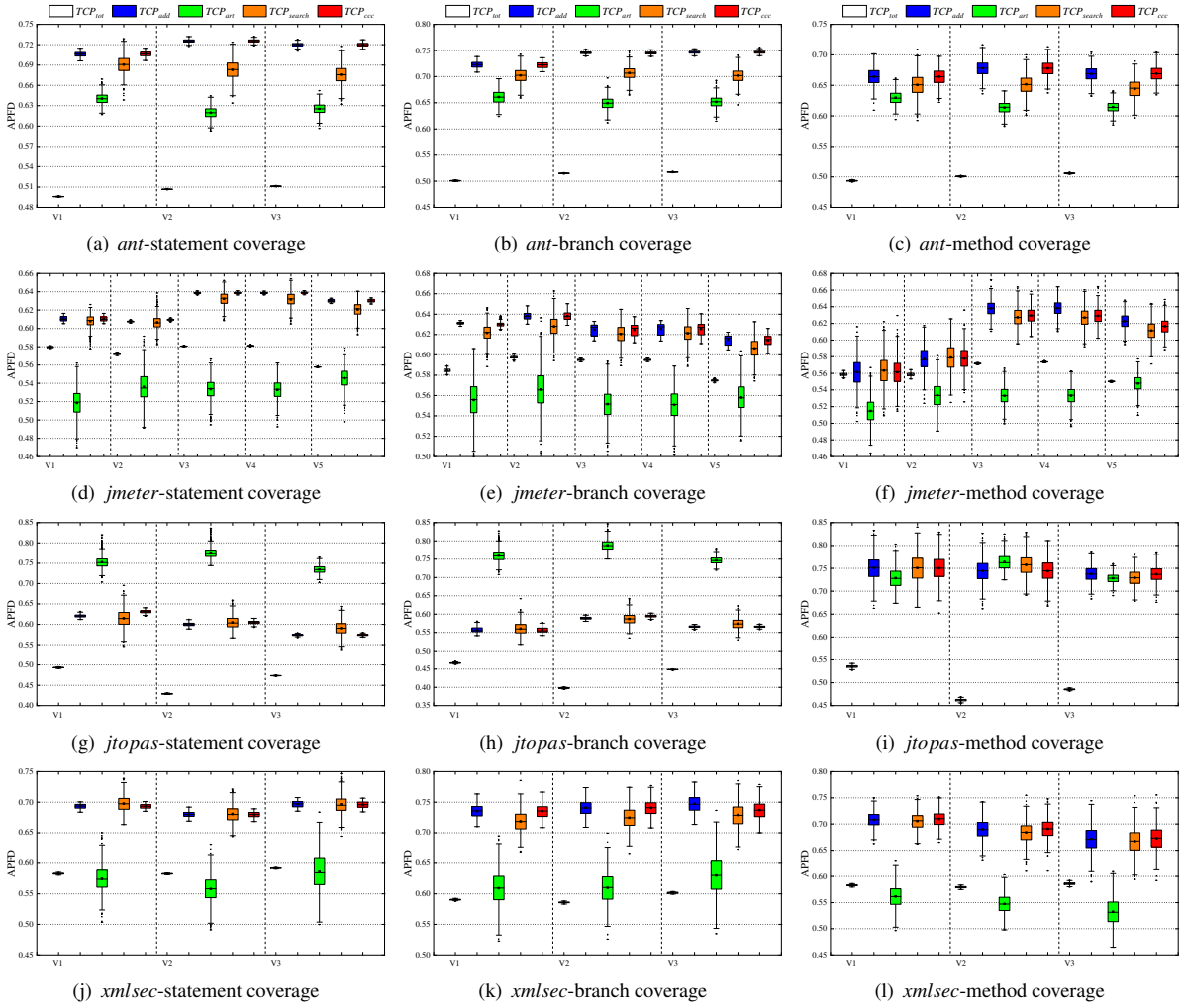
(5) Considering all Java programs: Overall, because all p-values are less than 0.05, and the relevant effect size $\hat{A}_{12}$ ranges from 0.58 to 0.98, TCP_{ccc} performs better than TCP_{tot} and TCP_{art}. However, CCCP has very similar (or slightly worse) performance to TCP_{add} and TCP_{search}, with $\hat{A}_{12}$ values of either 0.49 or 0.50.

5.1.2. *APFD Results: Java Programs (Test-Method Level)*

Based on Figure 3 and Table 4, we have the following observations:

(1) Our proposed method achieves much higher mean and median APFD values than TCP_{tot} for all programs with all code coverage granularities, with the maximum differences reaching approximately 30%.

(2) CCCP has very similar performance to TCP_{add}, with their mean and median APFD differences at around



(a) *ant*-statement coverage (b) *ant*-branch coverage (c) *ant*-method coverage

(d) *jmeter*-statement coverage (e) *jmeter*-branch coverage (f) *jmeter*-method coverage

(g) *jtopas*-statement coverage (h) *jtopas*-branch coverage (i) *jtopas*-method coverage

(j) *xmlsec*-statement coverage (k) *xmlsec*-branch coverage (l) *xmlsec*-method coverage

Fig. 3: **Effectiveness: APFD** results for **Java programs** at the **test-method level**

1%.

(3) Other than for some versions of *jtopas*, CCCP has much better performance than TCP_{art}.

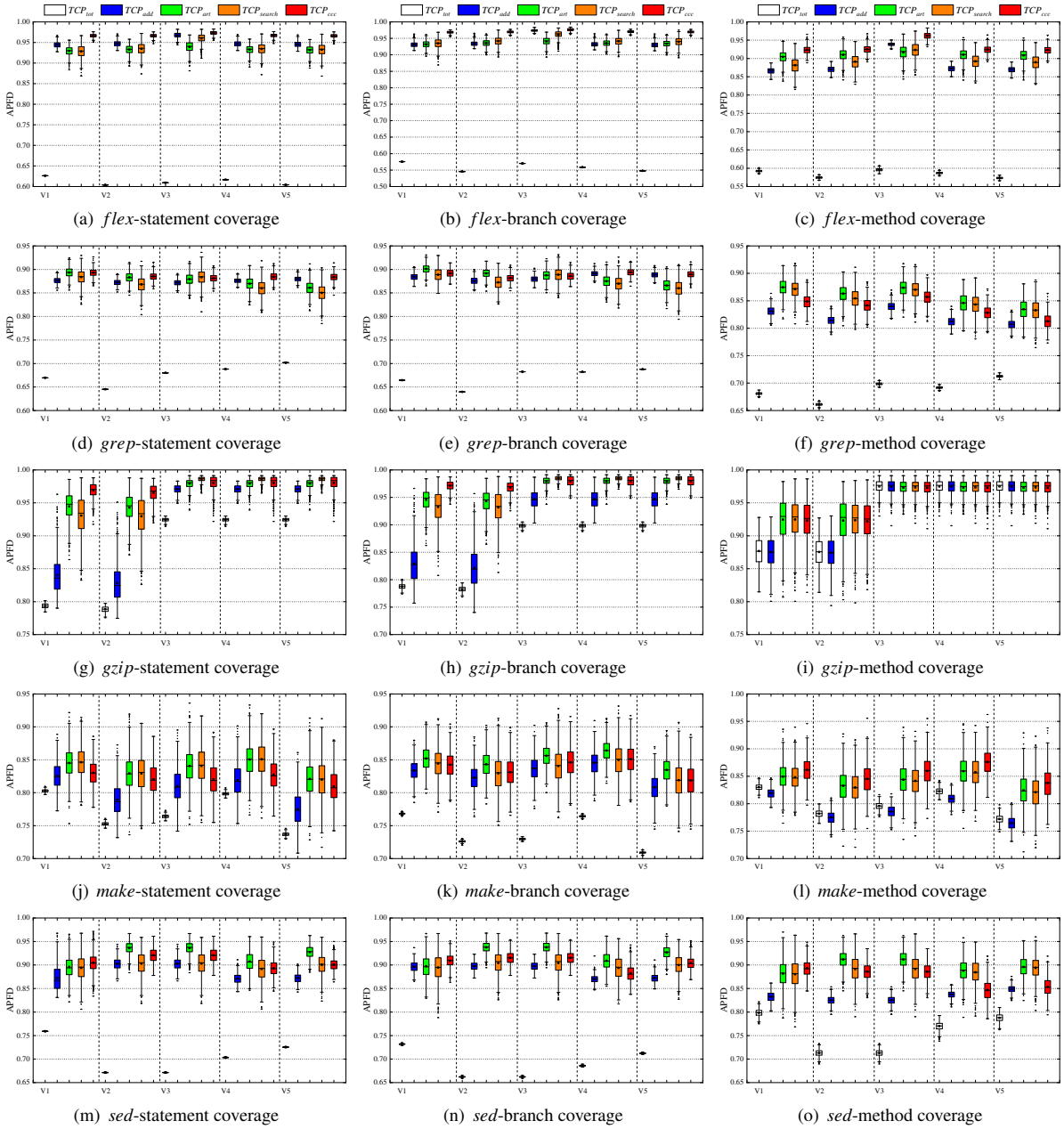
(4) Other than for a few cases (such as *jtopas_v*2 with method coverage, and *xmlsec_v*1 with statement coverage), CCCP usually has better performance than TCP_{search}.

(5) Overall, the statistical analysis supports the above box plots observations. Looking at all Java programs together, CCCP is, on the whole, better than TCP_{tot}, TCP_{art}, and TCP_{search}: The *p*-values are all less than 0.05, indicating that their differences are significant; and the effect size $\hat{A}_{12}$ values range from 0.54 to 0.97, which means that TCP_{ccc} is better than the other three RTCP techniques. Finally, while the *p*-values for comparisons between TCP_{ccc} and TCP_{add} are less than 0.05 (which means that the differences are significant), the $\hat{A}_{12}$ values range from 0.49 to 0.51, indicating that they are very similar.

5.1.3. APFD Results: C Subject Programs

Based on Figure 4 and Table 5, we have the following observations:

(1) Our proposed CCCP approach has much better performance than TCP_{tot} and TCP_{add}, for all programs and code coverage granularities, except for *gzip* with method coverage (for which TCP_{ccc} has very similar, or better performance). The maximum difference in mean and median APFD values between TCP_{ccc} and TCP_{tot} is more than 40%; while between TCP_{ccc} and TCP_{add}, it is



(a) *flex*-statement coverage (b) *flex*-branch coverage (c) *flex*-method coverage

(d) *grep*-statement coverage (e) *grep*-branch coverage (f) *grep*-method coverage

(g) *gzip*-statement coverage (h) *gzip*-branch coverage (i) *gzip*-method coverage

(j) *make*-statement coverage (k) *make*-branch coverage (l) *make*-method coverage

(m) *sed*-statement coverage (n) *sed*-branch coverage (o) *sed*-method coverage

Fig. 4: **Effectiveness: APFD** results for **C programs**

about 10%.

(2) TCP_{ccc} has similar or better APFD performance than TCP_{art} and TCP_{search} for some subject programs (such as *flex* and *gzip*, with all code coverage granularities), but also has slightly worse performance for some others (such as *grep* with method coverage and *sed* with statement coverage). However, the difference in mean and median APFD values between TCP_{ccc} and TCP_{art} or TCP_{search} is less than 5%.

(3) Overall, the statistical results support the box plot observations. All p values for the comparisons between TCP_{ccc} and TCP_{tot} or TCP_{add} are less than 0.05, indicating that their APFD scores are significantly different. The $\hat{A}_{12}$ values are also much greater than 0.50, rang-

Table 5: Statistical **effectiveness** comparisons of **APFD** for **C programs**. For a comparison between two methods TCP_{ccc} and M, where $M \in \{TCP_{tot}, TCP_{add}, TCP_{art}, TCP_{search}\}$, the symbol ✔ means that TCP_{ccc} is better (p-value is less than 0.05, and the effect size $\hat{A}_{12}(TCP_{ccc}, M)$ is greater than 0.50); the symbol ✖ means that M is better (the p-value is less than 0.05, and $\hat{A}_{12}(TCP_{ccc}, M)$ is less than 0.50); and the symbol ❍ means that there is no statistically significant difference between them (the p-value is greater than 0.05).

Program Name	Statement Coverage				Branch Coverage				Method Coverage			
	TCP_{tot}	TCP_{add}	TCP_{art}	TCP_{search}	TCP_{tot}	TCP_{add}	TCP_{art}	TCP_{search}	TCP_{tot}	TCP_{add}	TCP_{art}	TCP_{search}
flex_v1	✔(1.00)	✔(0.99)	✔(1.00)	✔(1.00)	✔(1.00)	✔(1.00)	✔(1.00)	✔(1.00)	✔(1.00)	✔(1.00)	✔(0.83)	✔(0.96)
flex_v2	✔(1.00)	✔(0.98)	✔(1.00)	✔(0.99)	✔(1.00)	✔(1.00)	✔(1.00)	✔(0.99)	✔(1.00)	✔(1.00)	✔(0.78)	✔(0.93)
flex_v3	✔(1.00)	✔(0.74)	✔(1.00)	✔(0.88)	✔(1.00)	✔(0.70)	✔(1.00)	✔(0.94)	✔(1.00)	✔(0.99)	✔(0.99)	✔(0.96)
flex_v4	✔(1.00)	✔(0.98)	✔(1.00)	✔(0.99)	✔(1.00)	✔(1.00)	✔(1.00)	✔(1.00)	✔(1.00)	✔(1.00)	✔(0.77)	✔(0.92)
flex_v5	✔(1.00)	✔(0.98)	✔(1.00)	✔(0.99)	✔(1.00)	✔(1.00)	✔(1.00)	✔(0.99)	✔(1.00)	✔(1.00)	✔(0.77)	✔(0.92)
flex	✔(1.00)	✔(0.89)	✔(1.00)	✔(0.94)	✔(1.00)	✔(0.86)	✔(1.00)	✔(0.95)	✔(1.00)	✔(0.86)	✔(0.80)	✔(0.88)
grep_v1	✔(1.00)	✔(0.94)	❍(0.49)	✔(0.70)	✔(1.00)	✔(0.75)	✖(0.23)	✔(0.57)	✔(1.00)	✔(0.86)	✖(0.10)	✖(0.15)
grep_v2	✔(1.00)	✔(0.89)	✔(0.55)	✔(0.82)	✔(1.00)	✔(0.70)	✖(0.21)	✔(0.70)	✔(1.00)	✔(0.96)	✖(0.14)	✖(0.28)
grep_v3	✔(1.00)	✔(0.82)	✔(0.56)	✖(0.44)	✔(1.00)	✔(0.72)	✖(0.45)	✖(0.43)	✔(1.00)	✔(0.86)	✖(0.20)	✖(0.27)
grep_v4	✔(1.00)	✔(0.79)	✔(0.81)	✔(0.90)	✔(1.00)	✔(0.61)	✔(0.89)	✔(0.91)	✔(1.00)	✔(0.84)	✖(0.21)	✖(0.25)
grep_v5	✔(1.00)	✔(0.65)	✔(0.91)	✔(0.95)	✔(1.00)	✔(0.54)	✔(0.93)	✔(0.93)	✔(1.00)	✔(0.63)	✖(0.18)	✖(0.21)
grep	✔(1.00)	✔(0.81)	✔(0.65)	✔(0.75)	✔(1.00)	✔(0.64)	✔(0.56)	✔(0.71)	✔(1.00)	✔(0.74)	✖(0.25)	✖(0.30)
gzip_v1	✔(1.00)	✔(1.00)	✔(0.85)	✔(0.89)	✔(1.00)	✔(1.00)	✔(0.89)	✔(0.90)	✔(0.88)	✔(0.88)	❍(0.48)	❍(0.48)
gzip_v2	✔(1.00)	✔(1.00)	✔(0.82)	✔(0.86)	✔(1.00)	✔(1.00)	✔(0.87)	✔(0.88)	✔(0.88)	✔(0.88)	❍(0.48)	❍(0.49)
gzip_v3	✔(1.00)	✔(0.80)	✔(0.60)	✖(0.39)	✔(1.00)	✔(0.95)	✔(0.53)	✖(0.36)	✖(0.46)	✖(0.46)	❍(0.50)	❍(0.49)
gzip_v4	✔(1.00)	✔(0.80)	✔(0.60)	✖(0.39)	✔(1.00)	✔(0.95)	✔(0.53)	✖(0.36)	✖(0.46)	✖(0.46)	❍(0.50)	❍(0.49)
gzip_v5	✔(1.00)	✔(0.80)	✔(0.60)	✖(0.39)	✔(1.00)	✔(0.95)	✔(0.53)	✖(0.36)	✖(0.46)	✖(0.46)	❍(0.50)	❍(0.49)
gzip	✔(1.00)	✔(0.79)	✔(0.62)	✔(0.52)	✔(1.00)	✔(0.95)	✔(0.61)	✔(0.52)	✔(0.55)	✔(0.56)	❍(0.50)	❍(0.49)
make_v1	✔(0.92)	✔(0.56)	✖(0.31)	✖(0.30)	✔(1.00)	✔(0.64)	✖(0.36)	✖(0.47)	✔(0.92)	✔(0.97)	✔(0.64)	✔(0.66)
make_v2	✔(1.00)	✔(0.81)	✖(0.40)	✖(0.40)	✔(1.00)	✔(0.61)	✖(0.35)	❍(0.52)	✔(0.99)	✔(0.99)	✔(0.62)	✔(0.65)
make_v3	✔(0.99)	✔(0.61)	✖(0.29)	✖(0.29)	✔(1.00)	✔(0.62)	✖(0.36)	✔(0.56)	✔(0.99)	✔(1.00)	✔(0.66)	✔(0.68)
make_v4	✔(0.89)	✔(0.60)	✖(0.26)	✖(0.26)	✔(1.00)	✔(0.58)	✖(0.32)	❍(0.51)	✔(0.99)	✔(1.00)	✔(0.68)	✔(0.69)
make_v5	✔(1.00)	✔(0.83)	✖(0.38)	✖(0.39)	✔(1.00)	✔(0.63)	✖(0.31)	❍(0.50)	✔(0.99)	✔(0.99)	✔(0.62)	✔(0.65)
make	✔(0.92)	✔(0.67)	✖(0.34)	✖(0.34)	✔(1.00)	✔(0.60)	✖(0.36)	❍(0.51)	✔(0.93)	✔(0.96)	✔(0.63)	✔(0.65)
sed_v1	✔(1.00)	✔(0.83)	✔(0.62)	✔(0.61)	✔(1.00)	✔(0.77)	✔(0.67)	✔(0.67)	✔(1.00)	✔(1.00)	✔(0.63)	✔(0.63)
sed_v2	✔(1.00)	✔(0.82)	✖(0.23)	✔(0.71)	✔(1.00)	✔(0.86)	✖(0.11)	✔(0.63)	✔(1.00)	✔(1.00)	✖(0.15)	✖(0.42)
sed_v3	✔(1.00)	✔(0.82)	✖(0.23)	✔(0.71)	✔(1.00)	✔(0.86)	✖(0.11)	✔(0.63)	✔(1.00)	✔(1.00)	✖(0.15)	✖(0.42)
sed_v4	✔(1.00)	✔(0.86)	✖(0.30)	❍(0.51)	✔(1.00)	✔(0.73)	✖(0.15)	✖(0.35)	✔(1.00)	✔(0.64)	✖(0.10)	✖(0.13)
sed_v5	✔(1.00)	✔(0.96)	✖(0.09)	❍(0.49)	✔(1.00)	✔(0.99)	✖(0.14)	✔(0.56)	✔(1.00)	✔(0.57)	✖(0.08)	✖(0.10)
sed	✔(1.00)	✔(0.78)	✖(0.33)	✔(0.60)	✔(1.00)	✔(0.78)	✖(0.28)	✔(0.57)	✔(1.00)	✔(0.89)	✖(0.25)	✖(0.35)
All C Programs	✔(0.93)	✔(0.64)	✔(0.54)	✔(0.56)	✔(0.95)	✔(0.65)	✔(0.53)	✔(0.56)	✔(0.84)	✔(0.70)	✖(0.48)	✔(0.52)

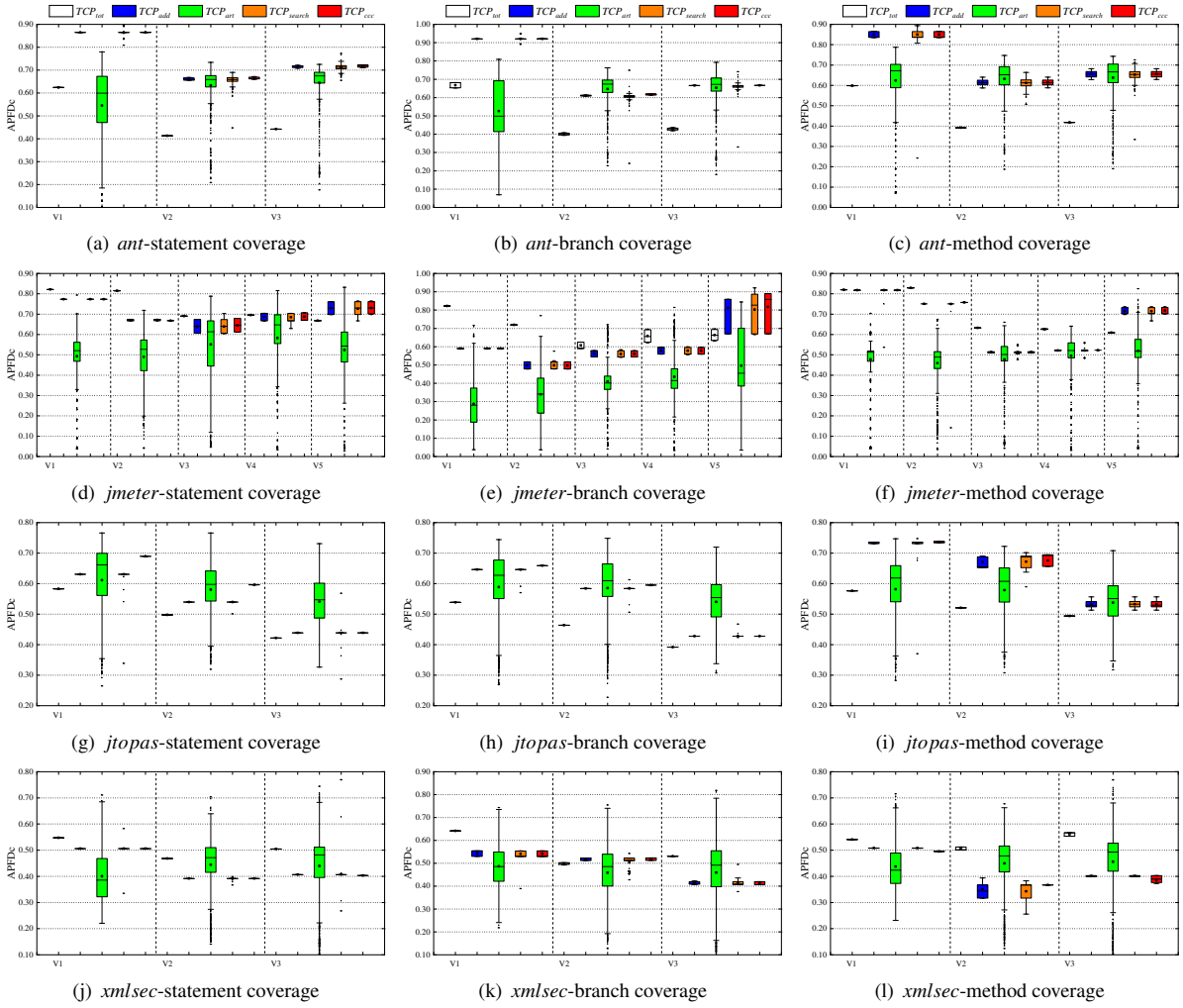
Table 6: An analysis of statistical **effectiveness** results of **APFD**. Each cell represents the total times of (✔), worse (✖), and (❍) for corresponding prioritization scenarios described in Tables 3 to 5.

Language	Status	Statement Coverage				Branch Coverage				Method Coverage				*Sum* ∑			
		TCP_{tot}	TCP_{add}	TCP_{art}	TCP_{search}	TCP_{tot}	TCP_{add}	TCP_{art}	TCP_{search}	TCP_{tot}	TCP_{add}	TCP_{art}	TCP_{search}	TCP_{tot}	TCP_{add}	TCP_{art}	TCP_{search}
Java (test-class)	✔	5	0	13	2	6	0	13	4	6	1	13	2	17	1	39	8
	✖	6	1	0	1	4	4	0	0	6	2	0	6	16	7	0	7
	❍	3	13	1	11	4	10	1	10	2	11	1	6	9	34	3	27
Java (test-method)	✔	14	6	11	10	14	1	11	12	14	1	13	10	42	8	35	32
	✖	0	1	3	2	0	5	3	2	0	3	1	2	0	9	7	6
	❍	0	7	0	2	0	8	0	0	0	10	0	2	0	25	0	4
C	✔	25	25	15	14	25	25	13	16	22	22	11	11	72	72	39	41
	✖	0	0	9	9	0	0	12	6	3	3	9	9	3	3	30	24
	❍	0	0	1	2	0	0	0	3	0	0	5	5	0	0	6	10
Sum ∑	✔	44	31	39	26	45	26	37	32	42	24	37	23	131	81	113	81
	✖	6	2	12	12	4	9	15	13	9	8	10	17	19	19	37	42
	❍	3	20	2	15	4	18	1	8	2	21	6	13	9	59	9	36

ing from 0.56 to 1.00 (except for the programs *gzip_v3*, *gzip_v4*, and *gzip_v5*, with method coverage). However, although all p values for the comparisons between TCP_{ccc} and TCP_{art} or TCP_{search} are also less than 0.05, their $\hat{A}_{12}$ values are much greater than 0.50 in some cases, but also much less than 0.50 in others. Nevertheless, considering all the C programs, not only does TCP_{ccc} have significantly different APFD values to the other four RTCP techniques, but it also has better performances overall (except for TCP_{art} with method coverage).

Table 6 summarizes the statistical results, presenting the total number of times TCP_{ccc} is better (✔), worse (✖), or not statistically different (❍), compared to the other RTCP techniques. Based on this table, we can answer RQ1 as follows:

1. When prioritizing Java test suites at the test-class level, TCP_{ccc} performs much better than TCP_{art}; similarly to TCP_{tot} and $TCP_{seaerch}$; and slightly worse than TCP_{add}.
2. When prioritizing Java test suites at the test-method level, TCP_{ccc} performs much better than



(a) *ant*-statement coverage (b) *ant*-branch coverage (c) *ant*-method coverage

(d) *jmeter*-statement coverage (e) *jmeter*-branch coverage (f) *jmeter*-method coverage

(g) *jtopas*-statement coverage (h) *jtopas*-branch coverage (i) *jtopas*-method coverage

(j) *xmlsec*-statement coverage (k) *xmlsec*-branch coverage (l) *xmlsec*-method coverage

Fig. 5: **Effectiveness: APFD$_c$** results for **Java programs** at the **test-class level**

TCP_{tot}, TCP_{art}, and TCP_{search}; and similarly to TCP_{add}.

3. When prioritizing C test suites, TCP_{ccc} performs much better than TCP_{tot}, TCP_{add}, and TCP_{search}; and slightly better than TCP_{art}.

In other words, as will be discussed in detail later (Sections 5.3 and 5.4), code coverage granularity and test case granularity may impact on the effectiveness of C-CCP, in terms of APFD. Nevertheless, the ratios of TCP_{ccc} performing better (✔) rather than worse (✖) than TCP_{tot}, TCP_{add}, TCP_{art}, and TCP_{search}, are: 6.89 (131/19), 4.26 (81/19), 3.05 (113/37), and 1.93 (81/42), respectively. In conclusion, overall, the proposed CCCP approach is more effective than the other four RTCP techniques, in terms of testing effectiveness, as measured by APFD.

5.2. *RQ2: Effectiveness of CCCP Measured by APFD$_c$*

Next, we provide the APFD$_c$ results for CCCP for different code coverage and test case granularities. Figures 5 to 7 show the APFD$_c$ results for the Java programs at the test-class level; at the test-method level; and the C programs, respectively. Each sub-figure in these figures has the program versions across the *x*-axis, and the APFD$_c$ values for the five RTCP techniques on the y-axis. Tables 7 to 9 present the corresponding statistical comparisons.

Table 7: Statistical **effectiveness** comparisons of **APFD$_c$** for **Java programs** at the **test-class level**. For a comparison between two methods TCP_{ccc} and M, where $M \in \{TCP_{tot}, TCP_{add}, TCP_{art}, TCP_{search}\}$, the symbol ✔ means that TCP_{ccc} is better (p-value is less than 0.05, and the effect size $\hat{A}_{12}(TCP_{ccc}, M)$ is greater than 0.50); the symbol ✖ means that M is better (the p-value is less than 0.05, and $\hat{A}_{12}(TCP_{ccc}, M)$ is less than 0.50); and the symbol ❍ means that there is no statistically significant difference between them (the p-value is greater than 0.05).

Program Name	Statement Coverage				Branch Coverage				Method Coverage			
	TCP_{tot}	TCP_{add}	TCP_{art}	TCP_{search}	TCP_{tot}	TCP_{add}	TCP_{art}	TCP_{search}	TCP_{tot}	TCP_{add}	TCP_{art}	TCP_{search}
ant_v1	✔(1.00)	✔(1.00)	✔(1.00)	✔(0.97)	✔(1.00)	✔(0.59)	✔(1.00)	✔(0.58)	✔(1.00)	✔(0.59)	✔(1.00)	✔(0.56)
ant_v2	✔(1.00)	✔(0.75)	✔(0.59)	✔(0.70)	✔(1.00)	✔(1.00)	✖(0.21)	✔(0.96)	✔(1.00)	❍(0.50)	✖(0.32)	✔(0.53)
ant_v3	✔(1.00)	✔(0.65)	✔(0.98)	✔(0.66)	✔(1.00)	✔(1.00)	✖(0.45)	✔(0.85)	✔(1.00)	❍(0.50)	✖(0.44)	✔(0.53)
ant	✔(1.00)	✔(0.60)	✔(0.85)	✔(0.59)	✔(0.83)	✔(0.62)	✔(0.61)	✔(0.60)	✔(0.98)	❍(0.51)	✔(0.59)	✔(0.52)
jmeter_v1	✖(0.00)	✖(0.00)	✔(1.00)	✖(0.00)	✖(0.00)	✔(0.56)	✔(0.96)	❍(0.52)	✖(0.00)	✔(0.67)	✔(1.00)	✔(0.64)
jmeter_v2	✖(0.00)	✖(0.14)	✔(0.99)	✖(0.12)	✖(0.00)	❍(0.49)	✔(0.86)	✖(0.47)	✖(0.00)	✔(1.00)	✔(1.00)	✔(1.00)
jmeter_v3	✖(0.00)	✔(0.66)	✔(0.69)	✔(0.67)	✖(0.00)	✔(0.54)	✔(0.86)	❍(0.48)	✖(0.00)	✖(0.47)	✔(0.63)	✖(0.46)
jmeter_v4	❍(0.49)	✔(0.73)	✔(0.78)	✔(0.75)	✖(0.00)	✔(0.53)	✔(0.81)	✖(0.46)	✖(0.00)	✔(0.70)	✔(0.58)	✔(0.72)
jmeter_v5	✔(1.00)	✔(0.61)	✔(0.92)	✔(0.64)	✔(0.92)	✔(0.62)	✔(0.92)	✔(0.55)	✔(1.00)	✔(0.57)	✔(0.98)	✔(0.62)
jmeter	✖(0.33)	❍(0.49)	✔(0.90)	❍(0.50)	✖(0.18)	❍(0.51)	✔(0.86)	❍(0.50)	✖(0.36)	✔(0.54)	✔(0.84)	✔(0.54)
jtopas_v1	✔(1.00)	✔(1.00)	✔(0.69)	✔(1.00)	✔(1.00)	✔(1.00)	✔(0.62)	✔(1.00)	✔(1.00)	✔(0.74)	✔(1.00)	✔(0.74)
jtopas_v2	✔(1.00)	✔(1.00)	❍(0.49)	✔(1.00)	✔(1.00)	✔(1.00)	✖(0.44)	✔(1.00)	✔(1.00)	✔(0.76)	✔(0.88)	✔(0.74)
jtopas_v3	✔(1.00)	❍(0.50)	✖(0.12)	❍(0.50)	✔(1.00)	❍(0.50)	✖(0.10)	❍(0.50)	✔(1.00)	❍(0.50)	✖(0.40)	❍(0.49)
jtopas	✔(0.78)	✔(0.61)	❍(0.50)	✔(0.61)	✔(0.78)	✔(0.61)	✖(0.47)	✔(0.61)	✔(0.87)	✔(0.56)	✔(0.73)	✔(0.55)
xmlsec_v1	✖(0.00)	❍(0.50)	✔(0.87)	❍(0.50)	✖(0.00)	✖(0.36)	✔(0.72)	❍(0.52)	✖(0.00)	✖(0.00)	✔(0.76)	✖(0.00)
xmlsec_v2	✖(0.00)	❍(0.50)	✖(0.22)	❍(0.50)	✔(1.00)	✔(0.75)	✔(0.64)	✔(0.81)	✖(0.00)	✔(0.66)	✖(0.16)	✔(0.76)
xmlsec_v3	✖(0.00)	✖(0.00)	✖(0.27)	✖(0.00)	✖(0.00)	✖(0.27)	✖(0.28)	✖(0.27)	✖(0.00)	✖(0.20)	✖(0.19)	✖(0.20)
xmlsec	✖(0.22)	✖(0.44)	✖(0.48)	✖(0.44)	✖(0.28)	❍(0.49)	✔(0.54)	❍(0.51)	✖(0.00)	✖(0.41)	✖(0.35)	✖(0.44)
All Java Programs	✔(0.58)	✔(0.52)	✔(0.68)	✔(0.53)	✔(0.53)	✔(0.52)	✔(0.65)	✔(0.52)	✔(0.58)	❍(0.50)	✔(0.67)	✔(0.51)

Table 8: Statistical **effectiveness** comparisons of **APFD$_c$** for **Java programs** at the **test-method level**. For a comparison between two methods TCP_{ccc} and M, where $M \in \{TCP_{tot}, TCP_{add}, TCP_{art}, TCP_{search}\}$, the symbol ✔ means that TCP_{ccc} is better (p-value is less than 0.05, and the effect size $\hat{A}_{12}(TCP_{ccc}, M)$ is greater than 0.50); the symbol ✖ means that M is better (the p-value is less than 0.05, and $\hat{A}_{12}(TCP_{ccc}, M)$ is less than 0.50); and the symbol ❍ means that there is no statistically significant difference between them (the p-value is greater than 0.05).

Program Name	Statement Coverage				Branch Coverage				Method Coverage			
	TCP_{tot}	TCP_{add}	TCP_{art}	TCP_{search}	TCP_{tot}	TCP_{add}	TCP_{art}	TCP_{search}	TCP_{tot}	TCP_{add}	TCP_{art}	TCP_{search}
ant_v1	✖(0.00)	✔(0.54)	✔(1.00)	✖(0.45)	✖(0.00)	❍(0.50)	✔(1.00)	✔(0.68)	✖(0.00)	✖(0.42)	✔(1.00)	❍(0.48)
ant_v2	✔(0.99)	❍(0.50)	✔(1.00)	✔(0.56)	✔(1.00)	❍(0.52)	✔(1.00)	✔(0.87)	✖(0.44)	❍(0.52)	✔(0.99)	✔(0.64)
ant_v3	✖(0.00)	❍(0.49)	✔(1.00)	✔(0.63)	✔(1.00)	❍(0.48)	✔(1.00)	✔(0.77)	✖(0.17)	❍(0.47)	✔(0.99)	✔(0.64)
ant	✖(0.40)	❍(0.51)	✔(1.00)	✔(0.54)	✔(0.63)	❍(0.50)	✔(1.00)	✔(0.76)	✖(0.24)	✖(0.47)	✔(0.99)	✔(0.59)
jmeter_v1	✔(1.00)	✔(0.53)	✔(1.00)	✔(0.63)	✔(1.00)	❍(0.50)	✔(0.93)	✔(0.80)	✔(0.76)	❍(0.50)	✔(0.63)	✔(0.53)
jmeter_v2	✔(1.00)	✔(0.53)	✔(1.00)	✔(0.65)	✔(1.00)	❍(0.49)	✔(0.90)	✔(0.82)	✔(1.00)	❍(0.50)	✔(0.88)	✔(0.56)
jmeter_v3	✔(1.00)	❍(0.52)	✔(1.00)	✖(0.47)	✔(1.00)	✖(0.44)	✔(0.99)	✔(0.66)	✔(1.00)	✖(0.45)	✔(1.00)	✖(0.47)
jmeter_v4	✔(1.00)	❍(0.51)	✔(1.00)	❍(0.48)	✔(1.00)	❍(0.49)	✔(1.00)	✔(0.69)	✔(1.00)	✖(0.41)	✔(1.00)	✖(0.44)
jmeter_v5	✔(1.00)	❍(0.50)	✔(1.00)	✔(0.61)	✔(1.00)	✖(0.30)	✔(1.00)	✔(0.77)	✔(1.00)	❍(0.48)	✔(0.86)	✖(0.41)
jmeter	✔(1.00)	❍(0.51)	✔(1.00)	✔(0.57)	✔(1.00)	✖(0.46)	✔(0.97)	✔(0.69)	✔(0.95)	✖(0.48)	✔(0.85)	❍(0.49)
jtopas_v1	✔(1.00)	✔(1.00)	✖(0.00)	✔(0.81)	✔(1.00)	✖(0.37)	✖(0.00)	✖(0.36)	✔(1.00)	✔(0.53)	✖(0.31)	✖(0.47)
jtopas_v2	✔(1.00)	✔(0.95)	✖(0.00)	❍(0.51)	✔(1.00)	✔(0.64)	✖(0.00)	❍(0.50)	✔(1.00)	✔(0.53)	✖(0.04)	✖(0.33)
jtopas_v3	✔(1.00)	❍(0.51)	✖(0.00)	✖(0.18)	✔(1.00)	❍(0.50)	✖(0.00)	✖(0.28)	✔(1.00)	❍(0.49)	✖(0.05)	❍(0.51)
jtopas	✔(1.00)	✔(0.61)	✖(0.00)	❍(0.50)	✔(1.00)	❍(0.50)	✖(0.00)	✖(0.42)	✔(1.00)	✔(0.52)	✖(0.14)	✖(0.43)
xmlsec_v1	✔(1.00)	✖(0.27)	✖(0.38)	✖(0.38)	✔(1.00)	✔(0.53)	✖(0.36)	❍(0.52)	✔(1.00)	✔(0.53)	✔(0.69)	✔(0.56)
xmlsec_v2	✔(1.00)	❍(0.50)	✔(0.83)	✔(0.67)	✔(1.00)	✔(0.57)	✔(0.56)	✔(0.78)	✔(1.00)	❍(0.49)	✖(0.46)	✔(0.54)
xmlsec_v3	✔(1.00)	❍(0.50)	✖(0.12)	✔(0.61)	✔(1.00)	✖(0.45)	✖(0.01)	✖(0.14)	✔(1.00)	❍(0.50)	❍(0.48)	✔(0.55)
xmlsec	✔(1.00)	✖(0.47)	✖(0.46)	✔(0.54)	✔(0.83)	❍(0.51)	✖(0.34)	✔(0.53)	✔(1.00)	❍(0.50)	✔(0.53)	✔(0.54)
All Java Programs	✔(0.85)	❍(0.51)	✔(0.64)	✔(0.54)	✔(0.77)	❍(0.50)	✔(0.62)	✔(0.57)	✔(0.87)	✖(0.49)	✔(0.67)	✔(0.51)

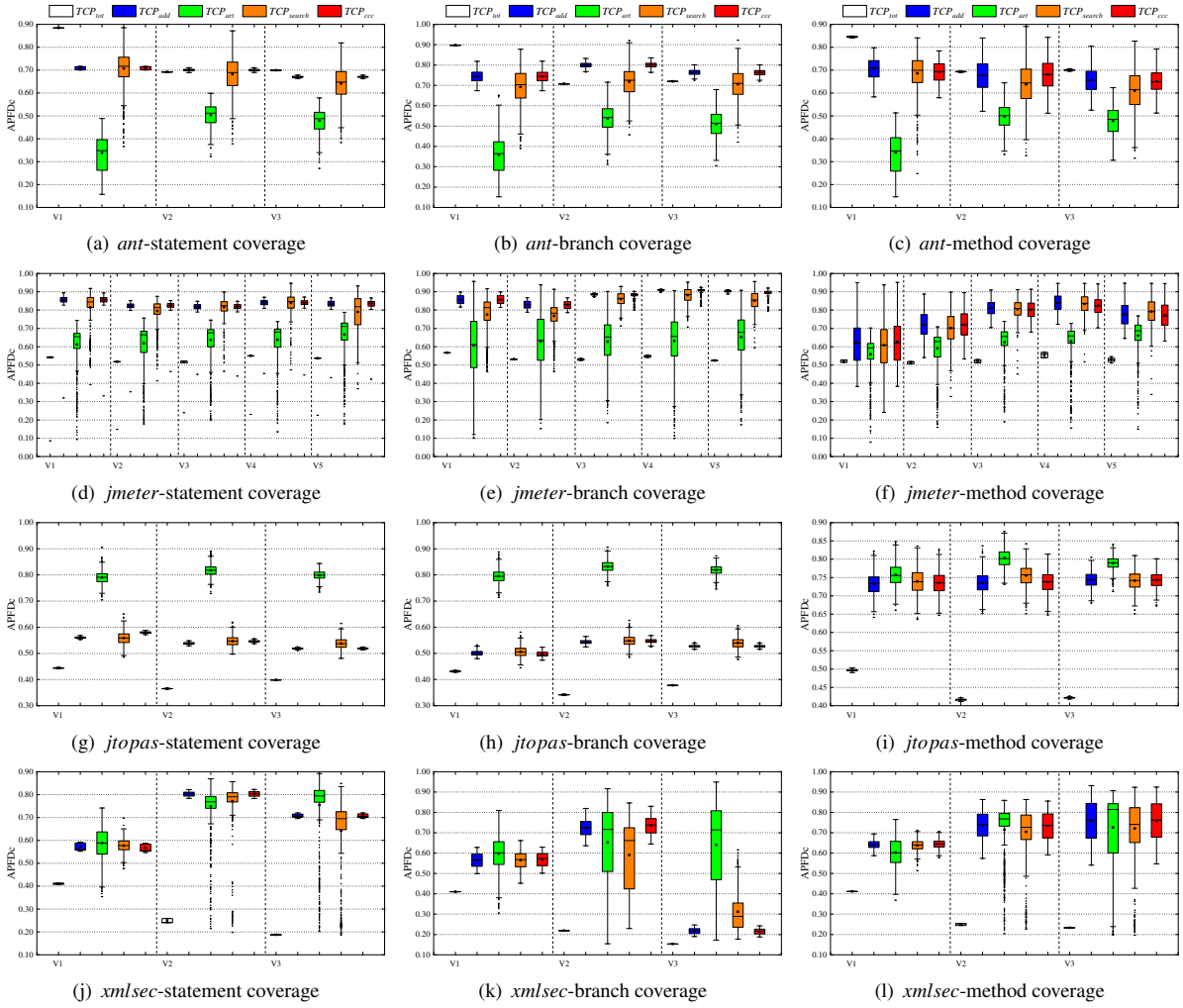
5.2.1. *APFD$_c$ Results: Java Programs (Test-Class Level)*

Based on Figure 5 and Table 7, we have the following observations:

(1) Compared with TCP_{tot}, TCP_{ccc} has much better APFD$_c$ results for the programs *ant* and *jtopas*, irrespective of program version and code coverage granularity, with the maximum difference between the mean and median values being up to about 30%. For the programs *jmeter* and *xmlsec*, however, TCP_{ccc} performs worse than TCP_{tot}, with a maximum APFD$_c$ difference of about 15%.

(2) Although TCP_{ccc} performs better than TCP_{art} in many cases (for example, with *jmeter*, for all code coverage granularities), it also sometimes performs worse (including with *ant_v2* and *ant_v3* for the branch and method coverage levels). Nevertheless, the TCP_{ccc} APFD$_c$ values have much lower variation than TCP_{art}.

(3) TCP_{ccc} has very similar performance to TCP_{add} and TCP_{search}, sometimes performing slightly better (for example, with *jtopas*, using statement coverage) or worse (such as with *xmlsec_v3*, for method coverage). The differences among the mean and median APFD$_c$ values for the three RTCP techniques are very small, at



(a) *ant*-statement coverage (b) *ant*-branch coverage (c) *ant*-method coverage

(d) *jmeter*-statement coverage (e) *jmeter*-branch coverage (f) *jmeter*-method coverage

(g) *jtopas*-statement coverage (h) *jtopas*-branch coverage (i) *jtopas*-method coverage

(j) *xmlsec*-statement coverage (k) *xmlsec*-branch coverage (l) *xmlsec*-method coverage

Fig. 6: **Effectiveness: APFD$_c$** results for **Java programs** at the **test-method level**

most, about 5%.

(4) Overall, the statistical analyses support the box plot observations. Looking at all Java programs together, TCP_{ccc} is better than TCP_{tot} and TCP_{art}, for all code coverage granularities: The p-values are much less than 0.05; and the $\hat{A}_{12}$ values range from 0.53 to 0.68. Furthermore, TCP_{ccc} performs similarly, or slightly better, compared with TCP_{add} and TCP_{search}, with the $\hat{A}_{12}$ values ranging from 0.50 to 0.53.

5.2.2. *APFD$_c$ Results: Java Programs (Test-Method Level)*

Based on Figure 6 and Table 8, we have the following observations:

(1) Apart from some cases with the program *ant* (for example, *ant_v*1), TCP_{ccc} has much better APFD$_c$ performance than TCP_{tot}, with the maximum mean and median differences reaching about 50%.

(2) TCP_{ccc} performs much better than TCP_{art} for the programs *ant* and *jmeter*, with the maximum mean and median APFD$_c$ differences being about 30%. In contrast, TCP_{art} performs much better than TCP_{ccc} for the program *jtopas*. For the program *xmlsec*, however, neither TCP_{art} nor TCP_{ccc} is always better: At branch coverage level, for example, TCP_{ccc} is better for version *v*2; TCP_{art} is better for version *v*3; and they have similar performance for version *v*1.

(3) The TCP_{ccc} and TCP_{add} APFD$_c$ distributions are very similar, in most cases, which means that, on the whole both techniques have very similar effectiveness

Table 9: Statistical **effectiveness** comparisons of **APFD$_c$** for **C programs**. For a comparison between two methods TCP_{ccc} and M, where $M \in \{TCP_{tot}, TCP_{add}, TCP_{art}, TCP_{search}\}$, the symbol ✔ means that TCP_{ccc} is better (p-value is less than 0.05, and the effect size $\hat{A}_{12}(TCP_{ccc}, M)$ is greater than 0.50); the symbol ✖ means that M is better (the p-value is less than 0.05, and $\hat{A}_{12}(TCP_{ccc}, M)$ is less than 0.50); and the symbol ❍ means that there is no statistically significant difference between them (the p-value is greater than 0.05).

Program Name	Statement Coverage				Branch Coverage				Method Coverage			
	TCP_{tot}	TCP_{add}	TCP_{art}	TCP_{search}	TCP_{tot}	TCP_{add}	TCP_{art}	TCP_{search}	TCP_{tot}	TCP_{add}	TCP_{art}	TCP_{search}
flex_v1	✔(1.00)	✔(0.99)	✔(1.00)	✔(1.00)	✔(1.00)	✔(1.00)	✔(1.00)	✔(1.00)	✔(1.00)	✔(1.00)	✔(0.82)	✔(0.96)
flex_v2	✔(1.00)	✔(0.98)	✔(1.00)	✔(0.99)	✔(1.00)	✔(1.00)	✔(1.00)	✔(0.99)	✔(1.00)	✔(1.00)	✔(0.76)	✔(0.93)
flex_v3	✔(1.00)	✔(0.75)	✔(1.00)	✔(0.88)	✔(1.00)	✔(0.71)	✔(1.00)	✔(0.94)	✔(1.00)	✔(0.99)	✔(0.99)	✔(0.96)
flex_v4	✔(1.00)	✔(0.99)	✔(1.00)	✔(0.99)	✔(1.00)	✔(1.00)	✔(1.00)	✔(1.00)	✔(1.00)	✔(1.00)	✔(0.75)	✔(0.92)
flex_v5	✔(1.00)	✔(0.98)	✔(1.00)	✔(0.99)	✔(1.00)	✔(1.00)	✔(1.00)	✔(1.00)	✔(1.00)	✔(1.00)	✔(0.75)	✔(0.92)
flex	✔(1.00)	✔(0.89)	✔(1.00)	✔(0.94)	✔(1.00)	✔(0.86)	✔(1.00)	✔(0.95)	✔(1.00)	✔(0.86)	✔(0.79)	✔(0.87)
grep_v1	✔(1.00)	✔(0.97)	✖(0.34)	✔(0.57)	✔(1.00)	✔(0.84)	✖(0.17)	❍(0.49)	✔(1.00)	✔(0.79)	✖(0.06)	✖(0.08)
grep_v2	✔(1.00)	✔(0.93)	✖(0.39)	✔(0.71)	✔(1.00)	✔(0.80)	✖(0.14)	✔(0.61)	✔(1.00)	✔(0.92)	✖(0.09)	✖(0.18)
grep_v3	✔(1.00)	✔(0.90)	✖(0.42)	✖(0.33)	✔(1.00)	✔(0.82)	✖(0.37)	✖(0.36)	✔(1.00)	✔(0.79)	✖(0.13)	✖(0.17)
grep_v4	✔(1.00)	✔(0.87)	✔(0.70)	✔(0.82)	✔(1.00)	✔(0.73)	✔(0.85)	✔(0.88)	✔(1.00)	✔(0.75)	✖(0.14)	✖(0.16)
grep_v5	✔(1.00)	✔(0.75)	✔(0.84)	✔(0.91)	✔(1.00)	✔(0.67)	✔(0.90)	✔(0.91)	✔(1.00)	❍(0.51)	✖(0.13)	✖(0.13)
grep	✔(1.00)	✔(0.88)	✔(0.54)	✔(0.66)	✔(1.00)	✔(0.74)	❍(0.50)	✔(0.66)	✔(1.00)	✔(0.68)	✖(0.19)	✖(0.22)
gzip_v1	✔(1.00)	✔(1.00)	✔(0.86)	✔(0.89)	✔(1.00)	✔(1.00)	✔(0.89)	✔(0.90)	✔(0.88)	✔(0.88)	❍(0.48)	❍(0.48)
gzip_v2	✔(1.00)	✔(1.00)	✔(0.82)	✔(0.86)	✔(1.00)	✔(1.00)	✔(0.87)	✔(0.88)	✔(0.88)	✔(0.88)	❍(0.48)	❍(0.49)
gzip_v3	✔(1.00)	✔(0.80)	✔(0.61)	✖(0.40)	✔(1.00)	✔(0.95)	✔(0.53)	✖(0.36)	✖(0.46)	✖(0.46)	❍(0.50)	❍(0.49)
gzip_v4	✔(1.00)	✔(0.80)	✔(0.61)	✖(0.40)	✔(1.00)	✔(0.95)	✔(0.53)	✖(0.36)	✖(0.46)	✖(0.46)	❍(0.50)	❍(0.49)
gzip_v5	✔(1.00)	✔(0.80)	✔(0.61)	✖(0.40)	✔(1.00)	✔(0.95)	✔(0.53)	✖(0.36)	✖(0.46)	✖(0.46)	❍(0.50)	❍(0.49)
gzip	✔(0.76)	✔(0.69)	✔(0.59)	✔(0.52)	✔(0.76)	✔(0.74)	✔(0.57)	✔(0.51)	✔(0.55)	✔(0.55)	❍(0.50)	❍(0.49)
make_v1	✔(0.98)	✖(0.44)	✖(0.23)	✖(0.19)	✔(1.00)	✔(0.70)	✖(0.23)	✖(0.30)	✔(1.00)	✔(1.00)	✔(0.79)	✔(0.80)
make_v2	✔(1.00)	✔(0.71)	✖(0.31)	✖(0.27)	✔(1.00)	✔(0.67)	✖(0.21)	✖(0.34)	✔(1.00)	✔(1.00)	✔(0.76)	✔(0.79)
make_v3	✔(1.00)	❍(0.50)	✖(0.22)	✖(0.19)	✔(1.00)	✔(0.68)	✖(0.23)	✖(0.39)	✔(1.00)	✔(1.00)	✔(0.79)	✔(0.81)
make_v4	✔(0.94)	❍(0.49)	✖(0.19)	✖(0.16)	✔(1.00)	✔(0.64)	✖(0.20)	✖(0.35)	✔(1.00)	✔(1.00)	✔(0.79)	✔(0.81)
make_v5	✔(1.00)	✔(0.73)	✖(0.29)	✖(0.26)	✔(1.00)	✔(0.69)	✖(0.18)	✖(0.32)	✔(1.00)	✔(1.00)	✔(0.76)	✔(0.79)
make	✔(0.98)	✔(0.56)	✖(0.30)	✖(0.27)	✔(1.00)	✔(0.63)	✖(0.29)	✖(0.37)	✔(1.00)	✔(1.00)	✔(0.73)	✔(0.75)
sed_v1	✔(1.00)	✔(0.85)	✔(0.57)	✔(0.64)	✔(1.00)	✔(0.83)	✔(0.61)	✔(0.68)	✔(1.00)	✔(1.00)	✔(0.68)	✔(0.71)
sed_v2	✔(1.00)	✔(0.86)	✖(0.18)	✔(0.73)	✔(1.00)	✔(0.91)	✖(0.08)	✔(0.65)	✔(1.00)	✔(1.00)	✖(0.19)	❍(0.52)
sed_v3	✔(1.00)	✔(0.86)	✖(0.18)	✔(0.73)	✔(1.00)	✔(0.91)	✖(0.08)	✔(0.65)	✔(1.00)	✔(1.00)	✖(0.19)	❍(0.52)
sed_v4	✔(1.00)	✔(0.90)	✖(0.24)	✔(0.53)	✔(1.00)	✔(0.82)	✖(0.12)	✖(0.37)	✔(1.00)	✔(0.76)	✖(0.13)	✖(0.19)
sed_v5	✔(1.00)	✔(0.98)	✖(0.06)	❍(0.51)	✔(1.00)	✔(0.99)	✖(0.10)	✔(0.57)	✔(1.00)	✔(0.73)	✖(0.10)	✖(0.15)
sed	✔(1.00)	✔(0.82)	✖(0.29)	✔(0.63)	✔(1.00)	✔(0.82)	✖(0.24)	✔(0.59)	✔(1.00)	✔(0.94)	✖(0.29)	✖(0.43)
All C Programs	✔(0.79)	✔(0.58)	✔(0.52)	✔(0.54)	✔(0.80)	✔(0.59)	✔(0.51)	✔(0.53)	✔(0.75)	✔(0.61)	✖(0.49)	✔(0.51)

(according to APFD$_c$).

(4) Although there are some large performance differences between TCP_{ccc} and TCP_{search} (such as for *ant_v2* and *xmlsec*, with branch coverage), overall, they have similar mean and median APFD$_c$ values. In most cases, TCP_{ccc} has lower variation in APFD$_c$ values than TCP_{search}.

(5) Overall, the statistical analyses support the box plot observations. Looking at all Java programs together, TCP_{ccc} is better than TCP_{tot} and TCP_{art}, with p-values much less than 0.05, and the $\hat{A}_{12}$ values ranging from 0.62 to 0.87. TCP_{ccc} is better than TCP_{search}, with the p-values less than 0.05, and the $\hat{A}_{12}$ values ranging from 0.51 to 0.57. Finally, TCP_{ccc} and TCP_{add} have very similar performance: The $\hat{A}_{12}$ values only range between 0.49 and 0.51; and the p-values are greater than 0.05 for statement and branch coverage, but less than 0.05 for method coverage.

5.2.3. APFD$_c$ Results: C Programs

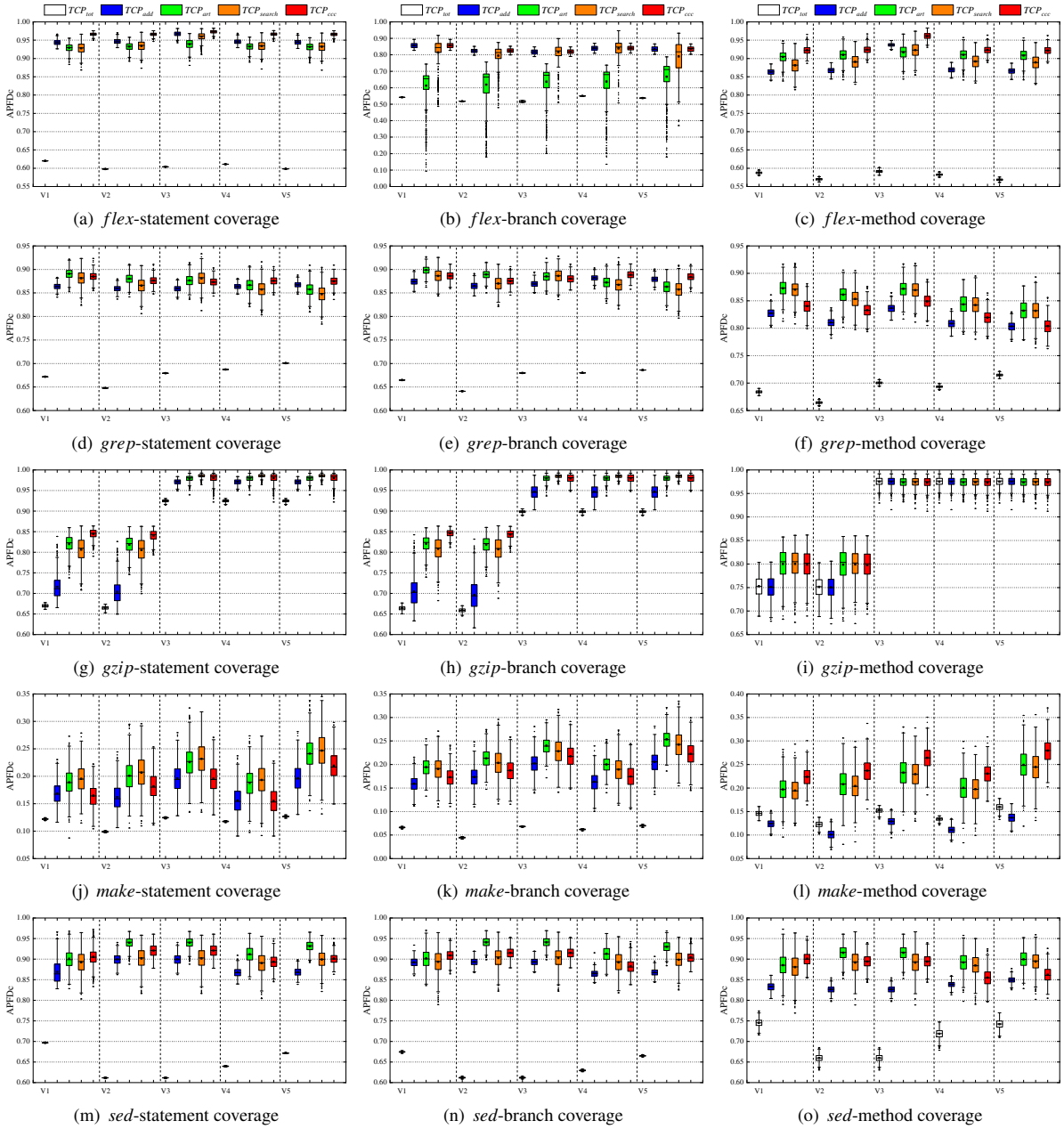
Based on Figure 7 and Table 9, we have the following observations:

(1) Except for some very few cases (such as *gzip_v3*, *gzip_v4*, and *gzip_v5*, with method coverage), TCP_{ccc} has much higher APFD$_c$ values than TCP_{tot} and TCP_{add}, with the maximum mean and median APFD$_c$ differences between TCP_{ccc} and TCP_{tot} reaching more than 35%; and between TCP_{ccc} and TCP_{add} being about 15%.

(2) TCP_{ccc} performs differently compared with TCP_{art} and TCP_{search} for different programs and different code coverage granularities: With the program *flex*, for example, for all versions and code coverage granularities, TCP_{ccc} is more effective; however, with *make*, for both statement and branch coverage, TCP_{art} and TCP_{search} are more effective.

(3) Overall, the statistical results support the box plot observations. Considering all C programs, the p values for all comparisons between TCP_{ccc} and TCP_{tot}, TCP_{add}, TCP_{art}, and TCP_{search} are less than 0.05, indicating that the APFD$_c$ scores are all significantly different. According to the effect size $\hat{A}_{12}$ values, TCP_{ccc} outperforms TCP_{tot} and TCP_{add}; and performs slightly better than TCP_{search} and TCP_{art} (except at the method coverage level).

Table 10 summarizes the statistical results, presenting the total number of times TCP_{ccc} is better (✔), worse (✖), or not statistically different (❍), compared to the other RTCP techniques. Based on this table, we can



(a) *flex*-statement coverage (b) *flex*-branch coverage (c) *flex*-method coverage

(d) *grep*-statement coverage (e) *grep*-branch coverage (f) *grep*-method coverage

(g) *gzip*-statement coverage (h) *gzip*-branch coverage (i) *gzip*-method coverage

(j) *make*-statement coverage (k) *make*-branch coverage (l) *make*-method coverage

(m) *sed*-statement coverage (n) *sed*-branch coverage (o) *sed*-method coverage

Fig. 7: **Effectiveness: APFD$_c$** results for **C programs**

answer RQ2 as follows:

1. When prioritizing Java test suites at the test-class level, TCP_{ccc} performs much better than TCP_{add}, TCP_{art} and TCP_{search}; and slightly better than TCP_{tot}.
2. When prioritizing Java test suites at the test-method level, TCP_{ccc} performs much better than TCP_{tot}, TCP_{art}, and TCP_{search}; and slightly better than TCP_{add}.
3. When prioritizing C test suites, TCP_{ccc} performs much better than TCP_{tot}, TCP_{add}, and TCP_{search}; and slightly better than TCP_{art}.

Similar to the APFD results, the CCCP APFD$_c$ performance is influenced by different factors, including the

Table 10: An analysis of statistical **effectiveness** results of **APFD$_c$**. Each cell represents the total times of (✔), worse (✖), and (❍) for corresponding prioritization scenarios described in Tables 7 to 9.

Language	Status	Statement Coverage				Branch Coverage				Method Coverage				*Sum* ∑			
		TCP_{tot}	TCP_{add}	TCP_{art}	TCP_{search}	TCP_{tot}	TCP_{add}	TCP_{art}	TCP_{search}	TCP_{tot}	TCP_{add}	TCP_{art}	TCP_{search}	TCP_{tot}	TCP_{add}	TCP_{art}	TCP_{search}
Java (test-class)	✔	7	8	10	8	8	10	9	7	7	8	9	10	22	26	28	25
	✖	6	3	3	3	6	2	5	3	7	3	5	3	19	8	13	9
	❍	1	3	1	3	0	2	0	4	0	3	0	1	1	8	1	8
Java (test-method)	✔	12	5	9	8	13	3	9	9	11	3	9	7	36	11	27	24
	✖	2	1	5	4	1	4	5	3	3	3	4	5	6	8	14	12
	❍	0	8	0	2	0	7	0	2	0	8	1	2	0	23	1	6
C	✔	25	22	13	15	25	25	13	14	22	21	11	11	72	68	37	40
	✖	0	1	12	9	0	0	12	10	3	3	9	7	3	4	33	26
	❍	0	2	0	1	0	0	0	1	0	1	5	7	0	3	5	9
Sum ∑	✔	44	35	32	31	45	46	38	31	40	32	29	28	129	113	99	90
	✖	8	5	20	16	4	7	6	22	13	9	18	15	25	21	44	53
	❍	1	13	1	6	4	0	9	0	0	12	6	10	5	25	16	16

type of test suite, and the code coverage granularity — both of which will be discussed in detail in the following two sections (Sections 5.3 and 5.4). Nevertheless, the ratios of TCP_{ccc} performing better (✔) rather than worse (✖) than TCP_{tot}, TCP_{add}, TCP_{art}, and TCP_{search}, are: 5.16 (129/25), 5.38 (113/21), 2.25 (99/44), and 1.70 (90/53), respectively. In conclusion, overall, the proposed CCCP approach is more effective than the four other RTCP techniques, in terms of testing effectiveness, as measured by APFD$_c$.

5.3. *RQ3: Impact of Code Coverage Granularity*

In this study, we examined three types of code coverage (statement, branch, and method). According to the APFD results (Figures 2 to 4) and APFD$_c$ results (Figures 5 to 7), in spite of some cases where the three types of code coverage provide very different APFD or APFD$_c$ results for CCCP, they do, overall, deliver comparable performance. This means that the choice of code coverage granularity may have little overall impact on the effectiveness of CCCP.

Figure 8 presents the APFD and APFD$_c$ results for the three types of code coverage, according to the subject programs' language or test suites (the language or test case granularity is shown on the x-axis; the APFD scores on the left y-axis; and the APFD$_c$ on the right y-axis). It can be observed that for C programs, statement and branch coverage are very considerable, but are more effective than method coverage. For Java programs, however, there is no best one among them, because they have similar APFD and APFD$_c$ values.

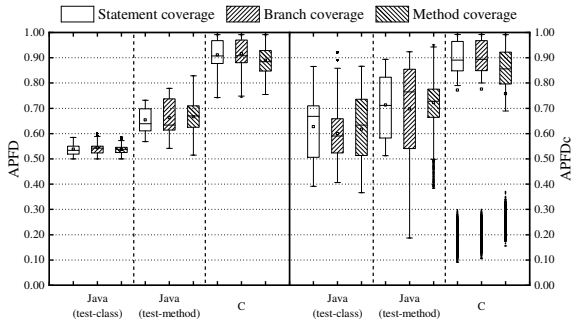


Fig. 8: **Effectiveness:** CCCP **APFD** and **APFD$_c$** results with different code coverage and test case granularities for **all programs**

Table 11 presents a comparison of the mean and median APFD and APFD$_c$ values, and also shows the p-values/effect size $\hat{A}_{12}$ for the different code coverage granularity comparisons. It can be seen from the table that the APFD and APFD$_c$ values are similar, with the maximum mean and median value differences being less than 3%, and less than 8%, respectively. According to the statistical comparisons, there is no single best code coverage type for CCCP, with each type sometimes achieving the best results. Nevertheless, branch coverage appears slightly more effective than statement and method coverage for CCCP.

In conclusion, the code coverage granularity may only provide a small impact on CCCP testing effectiveness, with branch coverage possibly performing slightly better than statement and method coverage.

5.4. *RQ4: Impact of Test Case Granularity*

To answer RQ4, we focus on the Java programs, each of which had two levels of test cases (the test-class and test-method levels). As can be seen in the comparisons between Figures 2 and 3, and between Figures 5 and 6, CCCP usually has significantly lower average APFD and APFD$_c$ values for prioritizing test cases at the test-class level than at the test-method level.

Considering all the Java programs, as can be seen in Table 11, the mean and median APFD and APFD$_c$ values at the test-method level are much higher than at the test-class level, regardless of code coverage granularity. One possible explanation for this is: Because a test case at the test-class level consists of a number of test cases

Table 11: Statistical **effectiveness** comparisons of **APFD** and **APFD$_c$** between different granularities for CCCP. Each cell in the **Mean**, **Median**, and **Comparison** columns represents the mean APFD or APFD$_c$ value, the median value, and the p-values/effect size $\hat{A}_{12}$ for the different code coverage granularity comparisons, respectively.

Metric	Language	Mean			Median			Comparison		
		Statement	Branch	Method	Statement	Branch	Method	Statement *vs* Branch	Statement *vs* Method	Branch *vs* Method
APFD	Java (test-class)	0.5385	0.5430	0.5369	0.5342	0.5420	0.5379	2.5E-60/0.44	6.1E-07/0.52	9.9E-106/0.58
	Java (test-method)	0.6542	0.6626	0.6664	0.6390	0.6339	0.6698	1.83E-40/0.45	1.4E-75/0.44	2.5E-07/0.48
	C	0.9115	0.9156	0.8901	0.9076	0.9077	0.8842	4.4E-18/0.48	0/0.62	0/0.64
APFD$_c$	Java (test-class)	0.6270	0.6002	0.6171	0.6679	0.5904	0.6336	2.3E-247/0.62	7.6E-08/0.52	1.1E-29/0.46
	Java (test-method)	0.7130	0.6972	0.7227	0.7112	0.7651	0.7279	2.3E-08/0.48	4.9E-02/0.49	5.0E-25/0.54
	C	0.7722	0.7758	0.7581	0.8909	0.8937	0.8565	1.1E-11/0.48	7.9E-222/0.58	1.4E-252/0.59

at the test-method level, prioritization at the test-method level may be more flexible, giving better fault detection effectiveness [10].

In conclusion, CCCP has better fault detection effectiveness when prioritizing test cases at the test-method level than at the test-class level.

5.5. *RQ5: CCCP Efficiency*

Table 12 presents the time overheads, in milliseconds, for the five RTCP techniques. The "**Comp.**" column presents the compilation times of the subject programs, and the "**Instr.**" column presents the instrumentation time (to collect the information of statement, branch, and method coverage). Apart from the first four columns, each cell in the table shows the prioritization time using each RTCP technique, for each program, presented as μ/σ (where μ is the mean time and σ is the standard deviation over the 1000 independent runs).

The Java programs had each version individually adapted to collect the code coverage information, with different versions using different test cases. Because of this, the execution time was collected for each Java program version. In contrast, each P_{V0} version of the C programs was compiled and instrumented to collect the code coverage information for each test case, and all program versions used the same test cases. Because of this, each C program version has the same compilation and instrumentation time. Furthermore, because all the studied RTCP techniques prioritized test cases after the coverage information was collected, they were all deemed to have the same compilation and instrumentation time for each version of each program.

Based on the time overheads, we have the following observations:

(1) As expected, the time overheads for all RTCP techniques (including CCCP) were lowest with method coverage, followed by branch, and then statement coverage, irrespective of test case type. The reason for this, as shown in Table 2, is that the number of methods is much lower than the number of branches, which in turn is lower than the number of statements; the related converted test cases are thus shorter, requiring less time to prioritize.

(2) It was also expected that (for the Java programs) prioritization at the test-method level would take longer than at the test-class level, regardless of code coverage granularity. The reason for this, again, relates to the number of test cases to be prioritized at the test-method level being more than at the test-class level.

(3) TCP_{ccc} requires much less time to prioritize test cases than TCP_{art} and TCP_{search}, and very similar time to TCP_{add}, irrespective of subject program, and code coverage and test case granularities. Also, because TCP_{tot} does not use feedback information during the prioritization process, it has much faster prioritization speeds than TCP_{ccc}.

In conclusion, TCP_{ccc} prioritizes test cases faster than TCP_{art} and TCP_{search}; has similar speed to TCP_{add}; but performs slower than TCP_{tot}.

5.6. *RQ6: CCCP Effectiveness with $\lambda = 2$*

To answer RQ6, this section briefly discusses the effectiveness of CCCP when $\lambda = 2$. Figure 9 shows the detailed APFD and APFDc results for the C programs, with the code coverage granularity on the x-axis, and the y-axis giving the APFD or APFDc scores. For ease of presentation, TCP_{ccc1} and TCP_{ccc2} denote CCCP with $\lambda = 1$ and $\lambda = 2$, respectively. Table 13 presents the statistical comparisons of the TCP_{ccc2} APFD and APFDc scores with those of the other five RTCP techniques: Each data cell shows the p-value/effect size $\hat{A}_{12}$ value.

Based on the experimental data, we have the following observations:

(1) TCP_{ccc2} has the higher mean and median APFD and APFD$_c$ values than TCP_{tot} and TCP_{add}, and better or similar to TCP_{art}, TCP_{search}, and TCP_{ccc1}, regardless of code coverage granularity.

(2) The statistical results confirm the box plot observations. All p-values are much less than 0.05, indicating a statistically significant difference between the

Table 12: **Efficiency:** Comparisons of execution costs in milliseconds for different RTCP techniques. The "**Comp.**" column presents the compilation times of the subject programs, and the "**Instr.**" column presents the instrumentation time (to collect the information of statement, branch, and method coverage). Apart from the first four columns, each cell in the table shows the prioritization time using each RTCP technique, for each program, presented as μ/σ (where μ is the mean time and σ is the standard deviation over the 1000 independent runs).

Language	Program	Time		Statement Coverage					Branch Coverage					Method Coverage					*Sum* ∑				
		Comp.	Instr.	TCP_{tot}	TCP_{add}	TCP_{art}	TCP_{search}	TCP_{ccc}	TCP_{tot}	TCP_{add}	TCP_{art}	TCP_{search}	TCP_{ccc}	TCP_{tot}	TCP_{add}	TCP_{art}	TCP_{search}	TCP_{ccc}	TCP_{tot}	TCP_{add}	TCP_{art}	TCP_{search}	TCP_{ccc}
Java (test-class)	*ant_v1*	10,386	62,671	0.3/0.5	6.4/0.5	149.6/13.2	8,629.9/100.4	5.0/0.7	0.1/0.3	1.3/0.5	40.9/5.2	1,899.6/59.1	1.4/0.6	0.0/0.2	1.0/0.2	29.1/4.3	1,491.3/94.5	0.9/0.3	0.4/–	8.7/–	219.6/–	12,020.8/–	7.3/–
	ant_v2	14,123	115,818	0.8/0.9	19.2/1.4	739.9/54.0	21,885.1/799.4	19.2/2.5	0.2/0.7	4.9/1.6	206.7/15.9	5,999.2/210.5	5.3/1.2	0.1/0.3	3.1/0.2	131.9/9.7	3,832.8/150.5	3.4/0.7	1.1/–	27.2/–	1,078.5/–	31,717.1/–	27.9/–
	ant_v3	36,126	116,404	0.8/0.5	19.2/1.9	740.2/52.3	22,076.9/2,134.4	19.2/2.3	0.2/0.4	4.9/1.1	204.3/16.4	6,120.5/234.1	5.3/0.7	0.1/0.4	3.1/0.9	132.5/10.4	3,903.5/172.4	3.5/0.6	1.1/–	27.2/–	1,077.0/–	32,100.9/–	28.0/–
	jmeter_v1	4,212	38,406	0.1/0.3	0.9/0.3	6.6/1.1	7,343.6/71.2	0.9/1.7	0.1/0.3	0.3/0.5	2.5/0.6	1,842.0/52.1	0.3/0.5	0.0/0.1	0.1/0.2	0.6/0.5	276.2/11.2	0.1/0.2	0.2/–	1.3/–	9.7/–	9,461.8/–	1.3/–
	jmeter_v2	4,737	40,469	0.2/0.7	1.1/0.2	8.8/1.4	7,740.5/76.5	1.0/0.1	0.1/0.2	0.4/0.5	3.3/0.8	1,992.8/49.5	0.4/0.5	0.0/0.1	0.1/0.3	0.8/0.5	316.4/7.5	0.1/0.2	0.3/–	1.6/–	12.9/–	10,049.7/–	1.5/–
	jmeter_v3	6,290	45,950	0.3/0.5	2.9/0.3	38.2/5.7	17,497.2/220.2	3.2/0.4	0.1/0.3	1.1/0.2	13.8/2.0	6,032.3/80.6	1.2/0.4	0.0/0.1	0.3/0.5	3.4/0.8	763.6/25.8	0.2/0.4	0.4/–	4.3/–	55.4/–	24,293.1/–	4.6/–
	jmeter_v4	13,395	41,501	0.1/0.3	1.1/0.6	21.0/2.6	4,865.3/57.4	1.3/0.5	0.0/0.1	0.2/0.4	4.5/0.7	613.8/20.1	0.3/0.8	0.0/0.1	0.2/0.4	4.1/0.7	516.8/14.7	0.2/0.4	0.1/–	1.5/–	29.6/–	5,995.9/–	1.8/–
	jmeter_v5	13,070	41,397	0.2/0.4	2.0/1.2	50.0/4.8	7,661.5/84.7	3.1/1.1	0.0/0.2	0.3/0.5	7.7/1.2	784.0/22.5	0.4/0.5	0.0/0.2	0.4/0.5	8.9/1.2	761.4/22.7	0.5/0.5	0.2/–	2.7/–	66.6/–	9,206.9/–	4.0/–
	jtopas_v1	15,327	22,317	0.2/0.5	3.1/1.1	83.5/7.2	9,028.7/129.9	4.0/0.3	0.0/0.2	0.4/0.5	11.9/1.5	867.4/25.9	0.6/0.7	0.1/0.3	0.7/0.5	17.7/1.8	1,050.6/33.9	0.8/0.4	0.3/–	4.2/–	113.1/–	10,946.7/–	5.4/–
	jtopas_v2	15,028	24,571	0.2/0.4	3.1/1.3	85.5/7.2	9,156.8/141.8	4.2/0.6	0.0/0.2	0.4/0.5	12.2/1.5	929.8/27.3	0.6/0.6	0.0/0.2	0.7/0.5	18.3/2.0	1,106.2/35.4	0.9/1.4	0.2/–	4.2/–	116.0/–	11,192.8/–	5.7/–
	jtopas_v3	16,879	35,646	0.2/0.4	4.0/0.8	122.3/9.9	9,996.7/258.5	5.4/2.2	0.0/0.2	0.6/0.5	14.9/1.7	1,021.1/37.7	0.7/0.4	0.1/0.2	0.9/0.4	25.4/2.4	1,249.2/51.8	1.0/0.1	0.3/–	5.5/–	162.6/–	12,267.0/–	7.1/–
	xmlsec_v1	9,722	12,570	0.2/0.4	1.6/0.5	21.0/2.3	6,781.8/74.3	1.3/0.5	0.0/0.2	0.5/0.5	4.9/1.0	716.8/17.3	0.3/0.5	0.0/0.2	0.2/0.4	3.2/0.7	536.7/12.6	0.2/0.4	0.2/–	2.3/–	29.1/–	8,035.3/–	1.8/–
	xmlsec_v2	10,352	25,234	0.2/0.4	1.7/0.5	21.8/2.7	6,781.6/728.6	1.4/0.5	0.1/0.2	0.4/0.5	5.0/1.2	808.6/18.2	0.3/0.5	0.0/0.2	0.2/0.4	3.1/0.7	512.6/12.6	0.2/0.4	0.3/–	2.3/–	29.9/–	8,102.8/–	1.9/–
	xmlsec_v3	10,276	31,700	0.2/0.7	1.3/0.4	14.5/1.9	4,926.8/986.1	1.1/0.8	0.0/0.2	0.4/0.5	3.4/0.9	643.7/16.3	0.3/0.4	0.0/0.2	0.2/0.4	2.0/0.5	409.8/10.8	0.1/0.4	0.2/–	1.9/–	19.9/–	5,980.3/–	1.5/–
Java (test-method)	*ant_v1*	–	–	1.1/0.6	82.7/3.2	7,895.2/363.0	25,552.6/685.0	67.1/4.5	0.4/0.8	17.9/2.4	723.8/42.6	5,950.3/400.1	19.3/3.1	0.3/1.0	15.1/1.9	1,456.9/70.7	5,055.7/339.3	13.0/2.0	1.8/–	115.7/–	10,075.9/–	36,558.6/–	99.4/–
	ant_v2	–	–	2.7/1.3	308.3/5.6	46,943.6/2,484.2	56,464.7/12,834.1	287.1/10.0	0.9/1.6	77.3/1.5	2,307.4/151.0	17,620.8/1,007.5	77.9/5.9	0.5/1.1	56.6/2.5	8,740.3/309.3	13,052.6/755.0	52.7/3.9	4.1/–	442.2/–	57,991.3/–	87,138.1/–	417.7/–
	ant_v3	–	–	2.8/1.5	303.8/5.2	46,487.4/1,950.7	56,055.4/12,828.2	284.0/8.9	0.8/1.0	77.0/1.4	2,284.0/146.0	11,463.7/2,429.0	76.8/2.7	0.5/1.0	55.0/0.9	8,591.8/308.4	9,456.1/1,669.7	52.0/2.3	4.1/–	435.8/–	57,363.2/–	76,975.2/–	412.8/–
	jmeter_v1	–	–	1.0/1.0	63.4/1.8	10,945.7/3,639.1	48,223.8/3,045.9	71.9/3.3	0.4/0.7	23.7/1.1	2,912.9/109.8	16,169.4/880.2	26.5/1.7	0.1/0.3	6.3/0.5	727.7/29.2	2,697.8/355.8	5.5/0.5	1.5/–	93.4/–	14,586.3/–	67,091.0/–	103.9/–
	jmeter_v2	–	–	1.0/0.5	66.7/1.8	11,800.5/3,838.7	50,776.5/3,564.1	75.3/3.0	0.4/0.5	25.0/1.5	3,061.5/128.3	17,139.4/1,003.3	28.2/1.6	0.1/0.3	6.6/0.5	792.7/31.6	3,122.9/341.7	6.0/0.3	1.5/–	98.3/–	15,654.7/–	71,038.8/–	109.5/–
	jmeter_v3	–	–	2.7/2.1	263.0/4.6	67,417.5/24,856.1	115,587.2/31,331.9	360.8/10.2	1.0/0.6	98.2/2.7	35,013.6/7,080.4	47,254.9/5,475.1	132.7/4.4	0.3/0.4	28.6/0.7	9,556.6/494.8	15,477.5/1,017.4	27.9/2.0	4.0/–	389.8/–	111,987.7/–	178,319.6/–	521.4/–
	jmeter_v4	–	–	0.2/0.4	7.8/0.4	854.6/94.2	18,956.3/3,208.7	9.3/1.7	0.1/0.2	1.7/0.5	62.3/16.8	3,388.9/805.1	2.1/0.2	0.0/0.2	1.4/0.5	168.0/33.4	4,024.7/564.2	1.7/0.5	0.3/–	10.9/–	1,084.9/–	26,369.9/–	13.1/–
	jmeter_v5	–	–	0.5/1.3	18.4/1.4	2,631.4/200.2	11,459.0/2,003.0	22.6/1.8	0.1/0.3	2.8/0.4	114.2/26.7	5,524.5/539.1	3.5/0.8	0.1/0.3	3.1/0.4	455.9/64.2	1,829.2/57.9	3.8/1.1	0.7/–	24.3/–	3,201.5/–	18,812.7/–	29.9/–
	jtopas_v1	–	–	0.6/0.5	23.5/0.6	3,408.0/273.4	17,396.6/702.3	27.3/2.7	0.1/0.3	3.3/0.5	144.5/34.2	1,717.5/73.2	4.2/0.6	0.1/0.4	4.9/0.7	721.3/91.1	2,489.7/110.4	5.4/1.7	0.8/–	31.7/–	4,273.8/–	21,603.8/–	36.9/–
	jtopas_v2	–	–	0.7/0.7	24.4/2.2	3,519.3/289.1	17,671.1/705.3	27.9/2.6	0.1/0.3	3.3/0.8	148.8/34.6	1,910.7/84.7	4.3/1.4	0.2/0.4	4.9/0.4	746.8/96.4	2,774.4/108.2	5.4/0.8	1.0/–	32.6/–	4,414.9/–	22,356.2/–	37.6/–
	jtopas_v3	–	–	0.7/1.0	28.9/0.7	4,476.6/352.7	20,455.8/577.5	34.0/3.2	0.1/0.3	3.9/1.0	182.5/37.9	2,091.8/100.0	5.0/0.9	0.2/0.9	5.9/0.9	922.1/112.4	3,105.3/129.6	6.5/0.9	1.0/–	38.7/–	5,581.2/–	25,652.9/–	45.5/–
	xmlsec_v1	–	–	0.9/1.0	36.8/2.1	5,623.0/413.0	16,075.6/468.9	34.5/3.3	0.2/0.5	8.8/0.7	282.4/55.3	2,172.4/111.3	8.4/1.4	0.1/0.3	5.6/1.2	828.4/102.2	2,235.5/78.4	5.3/1.0	1.2/–	51.2/–	6,733.8/–	20,483.5/–	48.2/–
	xmlsec_v2	–	–	0.9/0.6	41.2/2.3	6,438.4/443.9	13,324.8/3,148.0	38.1/2.9	0.3/0.6	10.9/0.6	356.0/60.1	2,581.1/122.9	9.8/1.6	0.1/0.3	6.1/0.9	903.5/105.9	2,149.4/73.5	5.8/1.5	1.3/–	58.2/–	7,697.9/–	18,055.3/–	53.7/–
	xmlsec_v3	–	–	0.9/1.3	33.7/2.2	4,257.9/734.8	10,055.2/2,334.9	29.7/2.2	0.3/0.4	9.0/1.3	265.0/50.0	2,113.3/107.5	8.0/1.6	0.1/0.6	4.5/0.5	626.2/78.5	1,322.8/202.3	4.2/1.1	1.3/–	47.2/–	5,149.1/–	13,491.3/–	41.9/–
C	*flex*	401	10,075	7.8/4.0	482.9/14.4	6,746.2/166.5	4,308.3/155.7	503.6/11.4	4.6/3.7	304.7/12.4	5,805.9/165.7	3,855.0/160.9	260.2/8.0	1.2/3.4	72.9/2.8	402.0/23.9	2,871.9/63.9	28.9/2.6	13.6/–	860.5/–	12,954.1/–	11,035.2/–	792.7/–
	grep	1,833	8,555	4.4/3.6	235.3/6.7	3,547.3/123.2	2,874.9/63.0	216.2/7.5	3.7/3.6	217.0/6.7	4,527.4/122.5	2,966.2/78.3	172.6/6.8	0.8/0.8	48.8/1.1	226.6/9.1	2,255.3/37.9	17.6/1.9	8.9/–	501.1/–	8,301.3/–	8,096.4/–	406.4/–
	gzip	308	1,882	0.9/0.7	13.6/0.6	70.2/4.4	517.1/10.0	16.5/1.9	0.6/0.7	9.4/0.5	50.2/3.6	460.9/10.1	10.2/1.4	0.3/0.7	3.5/0.5	7.4/1.2	368.5/9.5	2.3/0.8	1.8/–	26.5/–	127.8/–	1,346.5/–	29.0/–
	make	1,483	8,520	1.8/0.9	21.8/0.7	116.3/8.1	783.3/16.8	28.9/2.2	1.3/1.0	14.9/0.3	120.2/7.9	604.9/12.6	18.5/1.9	0.3/0.8	2.2/0.4	5.9/1.1	228.6/6.5	2.5/0.6	3.4/–	38.9/–	242.4/–	1,616.8/–	49.9/–
	sed	837	2,956	1.7/1.1	68.2/1.2	1,049.5/36.5	1,597.8/32.0	60.8/3.7	1.1/1.0	48.7/1.6	849.0/32.7	1,499.7/28.5	31.6/2.6	0.5/0.9	20.6/0.6	87.8/4.4	1,324.7/27.5	7.3/1.1	3.3/–	137.5/–	1,986.3/–	4,422.2/–	99.7/–
Sum ∑				37.3/–	2,192.0/–	236,331.5/–	632,508.4/–	2,265.9/–	17.4/–	973.6/–	59,747.6/–	176,757.0/–	917.2/–	6.2/–	363.8/–	36,348.9/–	92,569.7/–	265.9/–	60.9/–	3,529.4/–	332,428.0/–	901,835.1/–	3,449.0/–

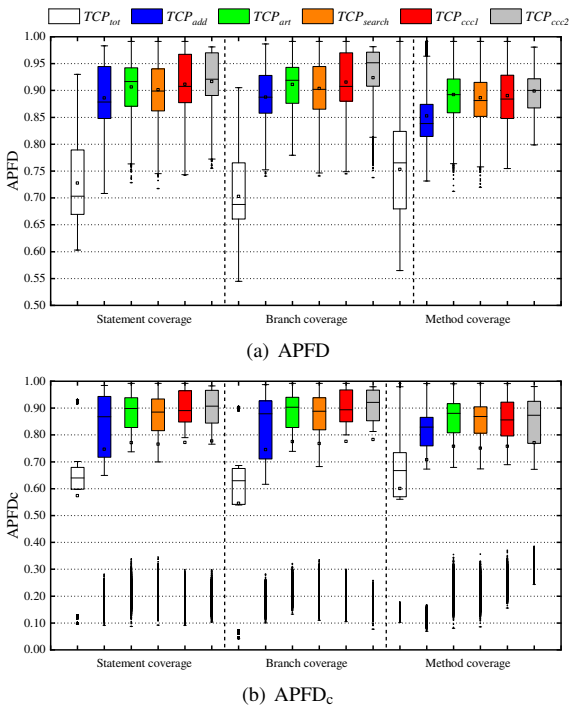
Table 13: Statistical **effectiveness** comparisons of **APFD** and **APFD$_c$** between CCCP with $\lambda = 2$ and the other five RTCP techniques for **all C programs**

Metric	Comparison	Statement	Branch	Method
APFD	vs TCP_{tot}	0/0.94	0/0.97	0/0.85
	vs TCP_{add}	0/0.66	0/0.71	0/0.77
	vs TCP_{art}	2.3E-149/0.57	0/0.60	6.3E-55/0.54
	vs TCP_{search}	8.4E-262/0.58	0/0.62	2.2E-253/0.59
	vs TCP_{ccc1}	3.0E-262/0.53	2.0E-80/0.55	4.8E-103/0.56
APFD$_c$	vs TCP_{tot}	0/0.79	0/0.82	0/0.74
	vs TCP_{add}	0/0.60	0/0.65	0/0.62
	vs TCP_{art}	5.0E-68/0.55	4.0E-158/0.57	1.4E-10/0.52
	vs TCP_{search}	2.5E-135/0.56	4.8E-237/0.58	1.4E-59/0.54
	vs TCP_{ccc1}	1.4E-40/0.53	1.5E-66/0.54	1.4E-22/0.53

TCP_{ccc2} and each of other five RTCP techniques, regardless of APFD and APFD$_c$ values. The $\hat{A}_{12}$ results also show TCP_{ccc2} to outperform TCP_{tot}, TCP_{add}, TCP_{art}, TCP_{search}, and TCP_{ccc1}, with probabilities ranging from 74% to 97%, 62% to 77%, 52% to 60%, 54% to 62%, and 53% to 56%, respectively.

These observations partly confirm our hypothesis about the performance of CCCP: As the λ for unit combination increases, the testing information for guiding prioritization is greater, which may will result in improved performance.

Finally, regarding the prioritization time: TCP_{ccc2} requires about 351073.3, 159881.4, and 501.8 milliseconds for the C programs when using statement, branch and method coverage, respectively. This low prioritization time of 501.8 milliseconds for TCP_{ccc2} with method coverage is less than the prioritization time for TCP_{add} using statement or branch coverage (2192.0 and 973.6 milliseconds, respectively). As shown in Figure 9, TCP_{ccc2} with method coverage has comparable fault detection effectiveness to TCP_{add} with statement or branch coverage. Because method coverage is usually much less expensive to achieve than statement or branch coverage, TCP_{ccc2} with method coverage should be a better choice than TCP_{add} with statement or branch coverage. Furthermore, method-level coverage — which is the most natural for projects that are large in scale and high in complexity — has greater potential practical application than statement and branch criteria, making TCP_{ccc2} with method coverage more feasible (2-wise code combinations coverage may incur much more complex calculations for statement and branch coverage than for method coverage).



(a) APFD

(b) APFD$_c$

Fig. 9: **Effectiveness: APFD** and **APFD$_c$** results for **all C programs**

5.7. Practical Guidelines

Here, we present some practical guidelines for how to choose the combination strength and code-coverage level for CCCP, under different testing scenarios:

(1) When testing resources are limited, it is (obviously) recommended that the lowest combination strength ($\lambda = 1$) be chosen for CCCP. This not only achieves better testing effectiveness than other prioritization techniques, but also has comparable testing speed to the *additional* test prioritization technique.

(2) When there are sufficient testing resources available, $\lambda = 2$ is recommended for CCCP, because of the higher fault detection rates it can deliver.

(3) If the system under test is large in scale and high in complexity, method coverage is recommended to be used for CCCP.

5.8. Threats to Validity

To facilitate the investigation of potential threats and to support the replication of experiments, we have made the relevant materials (including source code, subject programs, test suites, and mutants) available on our project website: https://github.com/huangrubing/CCCP/. Despite that, our study still face some threats to validity, listed as follows.

5.8.1. Internal Validity

The main threat to internal validity lies in the implementation of our experiment. First, the randomized

computations may affect the performance of CCCP: To address this, we repeated the prioritization process 1000 times and used statistical tests to assess the strategies. Second, the data structures used in the prioritization algorithms, and the faults in the source code, may introduce noise when evaluating the effectiveness and efficiency: To minimize these threats, we used data structures that were as similar as possible, and carefully reviewed all source code before conducting the experiment. Third, although we used the APFD and $APFD_c$ metrics, which have been extensively adopted to assess the performance of RTCP techniques, APFD only reflects the rate at which faults are detected, ignoring the time and space costs, and $APFD_c$ assumes that all faults have the same fault severity. To address this threat, our future work will involve additional metrics that can measure other practical performance aspects of prioritization strategies.

5.8.2. External Validity

All the programs used in the experiment were medium-sized, and written in C or Java, which means that the results may not be generalizable to programs written in other languages (such as C++ and C#) and of different sizes. To reduce this threat, other relevant programs will be adopted to evaluate the CCCP performance. Mutation testing has been argued to be an appropriate approach for assessing fault detection performance [27, 28, 29]. Mutation testing has also been used in recent RTCP research studies [17, 18, 19, 20]. However, Luo et al. [49] has highlighted the differences between real faults and mutants, explaining that the relative performances of RTCP techniques on mutants may not translate to similar relative performances with real faults. To address this threat, additional studies will be conducted to investigate the performance of RTCP on programs with real regression faults in the future.

6. Related Work

A considerable amount of research has been conducted into regression testing techniques with a goal of improving the testing performance. This includes test case prioritization [1, 50], reduction [51, 52] and selection [53, 54]. This Related Work section focuses on test case prioritization, which aims to detect faults as early as possible through the reordering of regression test cases [55, 56].

Prioritization Strategies. The most widely investigated prioritization strategies are the *total* and *additional* techniques [1]. Because existing greedy strategies may produce suboptimal results, Li et al. [2] translated the RTCP problem into a search problem and proposed several search-based algorithms, including a hill-climbing and genetic one. Motivated by random tie-breaking, Jiang et al. [3] applied adaptive random testing to RTCP and proposed a family of adaptive random test cases prioritization techniques that aim to select a test case with the greatest distance from already selected ones.

More recently, as the *total* strategy and the *additional* strategy can be seen as two extreme instances, Zhang et al. [10] proposed a basic and an extended model to unify the two strategies. Saha et al. [5] proposed an RTCP approach based on information retrieval without dynamic profiling or static analysis. Many existing RTCP approaches use code coverage to schedule the test cases, but do not consider the likely distribution of faults. To address this limitation, instead of traditional code coverage, Wang et al. [11] used the quality-aware code coverage calculated by code inspection techniques to guide prioritization process.

Coverage criteria. In terms of coverage criteria, structural coverage has been widely adopted in test case prioritization. In addition to statement [1], branch [3], method [10, 11], block [2] and modified condition/decision coverage [57], Elbaum et al. [30] proposed a fault-exposing-potential (FEP) criterion based on the probability of the test case detecting a fault. Recently, Chi et al. [58] used function call sequences, arguing that basic structural coverage may not be optimal for dynamic prioritization.

Empirical studies. A large number of empirical studies have been performed aiming to offer practical guidelines for using RTCP techniques.

In addition to studies on traditional dynamic test prioritization [1, 30, 59, 60], recently, Lu et al. [20] were the first to investigate how real-world software evolution impacts on the performance of prioritization strategies: They reported that source code changes have a low impact on the effectiveness of traditional dynamic techniques, but that the opposite was true when considering new tests in the process of evolution.

Citing a lack of comprehensive studies comparing static and dynamic test prioritization techniques, Luo et al. [17, 18] compared static RTCP techniques with dynamic ones. Henard et al. [19] compared white-box and back-box RTCP techniques.

7. Conclusions and Future work

In this paper, we have introduced a new coverage criterion that combines the concepts of code and combina-

tion coverage. Based on this, we proposed a new prioritization technique, *code combinations coverage based prioritization* (CCCP). Results from our empirical studies have demonstrated that CCCP with the lowest combination strength ($\lambda = 1$) can achieve better fault detection rates than four well-known, popular prioritization techniques (*total*, *additional*, *adaptive random*, and *search-based* test prioritization). CCCP was also found to have comparable testing efficiency to the *additional* test prioritization technique, while requiring much less time to prioritize test cases than the *adaptive random* and *search-based* techniques. The results also show that CCCP with a higher combination strength ($\lambda = 2$) can be more effective than all other prioritization techniques, in terms of both APFD and $APFD_c$.

Our future work will include examining more real-life programs to further investigate the performance of CCCP, including the impact of combination strengths. In this paper, we have only applied the concept of code combinations coverage to the traditional greedy prioritization strategy. It will be very interesting to examine new prioritization techniques based on code combinations coverage adopting other prioritization strategies such as search-based strategy.

Acknowledgements

We would like to thank the anonymous reviewers for their many constructive comments. We would also like to thank Christopher Henard for providing us the fault data for the five C subject programs. This work is supported by the National Natural Science Foundation of China under grant nos. 61502205, 61872167, and U1836116, the project funded by China Postdoctoral Science Foundation under grant no. 2019T120396, and the Senior Personnel Scientific Research Foundation of Jiangsu University under grant no. 14JDG039. This work is also in part supported by the Young Backbone Teacher Cultivation Project of Jiangsu University, and the Postgraduate Research & Practice Innovation Program of Jiangsu Province under grant no. KYCX19_1614.

References

[1] G. Rothermel, R. H. Untch, , M. J. Harrold, Test case prioritization: An empirical study, in: Proceedings the 15th IEEE International Conference on Software Maintenance (ICSM'99), 1999, pp. 179–188.

[2] Z. Li, M. Harman, R. M. Hierons, Search algorithms for regression test case prioritization, IEEE Transactions on Software Engineering 33 (4) (2007) 225–237.

[3] B. Jiang, Z. Zhang, W. K. Chan, T. Tse, Adaptive random test case prioritization, in: Proceedings of the 24th IEEE/ACM International Conference on Automated Software Engineering (ASE'09), 2009, pp. 233–244.

[4] H. Mei, D. Hao, L. Zhang, L. Zhang, J. Zhou, G. Rothermel, A static approach to prioritizing junit test cases, IEEE Transactions on Software Engineering 38 (6) (2012) 1258–1275.

[5] R. K. Saha, L. Zhang, S. Khurshid, D. E. Perry, An information retrieval approach for regression test prioritization based on program changes, in: Proceedings of the 37th IEEE/ACM IEEE International Conference on Software Engineering (ICSE'15), 2015, pp. 268–279.

[6] Y. Ledru, A. Petrenko, S. Boroday, Using string distances for test case prioritisation, in: Proceedings of the 24th IEEE/ACM International Conference on Automated Software Engineering (ASE'09), 2009, pp. 510–514.

[7] D. Hao, L. Zhang, L. Zang, Y. Wang, X. Wu, T. Xie, To be optimal or not in test-case prioritization, IEEE Transactions on Software Engineering 42 (5) (2016) 490–505.

[8] D. Hao, L. Zhang, L. Zhang, G. Rothermel, H. Mei, A unified test case prioritization approach, ACM Transactions on Software Engineering and Methodology 24 (2) (2014) 10:1–10:31.

[9] L. Zhang, J. Zhou, D. Hao, L. Zhang, H. Mei, Prioritizing JUnit test cases in absence of coverage information, in: Proceedings of the 25th IEEE International Conference on Software Maintenance (ICSM'09), 2009, pp. 19–28.

[10] L. Zhang, D. Hao, L. Zhang, G. Rothermel, H. Mei, Bridging the gap between the total and additional test-case prioritization strategies, in: Proceedings of the 2013 International Conference on Software Engineering (ICSE'13), 2013, pp. 192–201.

[11] S. Wang, J. Nam, L. Tan, QTEP: Quality-aware test case prioritization, in: Proceedings of the 11th Joint Meeting on Foundations of Software Engineering (ESEC/FSE'17), 2017, pp. 523–534.

[12] H. Zhu, P. A. V. Hall, J. H. R. May, Software unit test coverage and adequacy, ACM Computing Surveys 29 (4) (1997) 366–427.

[13] R. Huang, W. Sun, Y. Xu, H. Chen, D. Towey, X. Xia, A survey on adaptive random testing, IEEE Transactions on Software Engineering (2019). doi:10.1109/TSE.2019.2942921.

[14] C. Nie, H. Leung, A survey of combinatorial testing, ACM Computer Surveys 43 (2) (2011) 11:1–11:29.

[15] M. Grindal, B. Lindström, J. Offutt, S. F. Andler, An evaluation of combination strategies for test case selection, Empirical Software Engineering 11 (4) (2006) 583–611.

[16] Z. Zhang, J. Zhang, Characterizing failure-causing parameter interactions by adaptive testing, in: Proceedings of the 20th International Symposium on Software Testing and Analysis (ISSTA'11), 2011, pp. 331–341.

[17] Q. Luo, K. Moran, D. Poshyvanyk, A large-scale empirical comparison of static and dynamic test case prioritization techniques, in: Proceedings of the 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE'16), 2016, pp. 559–570.

[18] Q. Luo, K. Moran, L. Zhang, D. Poshyvanyk, How do static and dynamic test case prioritization techniques perform on modern software systems? An extensive study on GitHub projects, IEEE Transactions on Software Engineering 45 (11) (2019) 1054–1080.

[19] C. Henard, M. Papadakis, M. Harman, Y. Jia, Y. Le Traon, Comparing white-box and black-box test prioritization, in: Proceedings of the 38th IEEE/ACM International Conference on Software Engineering (ICSE'16), 2016, pp. 523–534.

[20] Y. Lu, Y. Lou, S. Cheng, L. Zhang, D. Hao, Y. Zhou, L. Zhang, How does regression test prioritization perform in real-world software evolution?, in: Proceedings of the 38th International

Conference on Software Engineering (ICSE'16), 2016, pp. 535–546.
[21] S. Elbaum, A. Malishevsky, G. Rothermel, Incorporating varying test costs and fault severities into test case prioritization, in: Proceedings of the 23rd International Conference on Software Engineering (ICSE'01), 2001, pp. 329–338.
[22] M. G. Epitropakis, S. Yoo, M. Harman, E. K. Burke, Empirical evaluation of pareto efficient multi-objective regression test case prioritisation, in: Proceedings of the 23rd International Symposium on Software Testing and Analysis (ISSTA'15), 2015, pp. 234–245.
[23] H. Do, S. Elbaum, G. Rothermel, Supporting controlled experimentation with testing techniques: An infrastructure and its potential impact, Empirical Software Engineering 10 (4) (2005) 405–435.
[24] Software-artifact Infrastructure Repository (SIR), https://sir.csc.ncsu.edu/portal/index.php.
[25] GNU FTP Server, http://ftp.gnu.org/.
[26] S. Eghbali, L. Tahvildari, Test case prioritization using lexicographical ordering, IEEE Transactions on Software Engineering 42 (12) (2016) 1178–1195.
[27] J. H. Andrews, L. C. Briand, Y. Labiche, Is mutation an appropriate tool for testing experiments?, in: Proceedings of the 27th International Conference on Software Engineering, (ICSE'05), 2005, pp. 402–411.
[28] H. Do, G. Rothermel, A controlled experiment assessing test case prioritization techniques via mutation faults, in: Proceedings of the 21st IEEE International Conference on Software Maintenance (ICSM'05), 2005, pp. 411–420.
[29] R. Just, D. Jalali, L. Inozemtseva, M. D. Ernst, R. Holmes, G. Fraser, Are mutants a valid substitute for real faults in software testing?, in: Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE'14), 2014, pp. 654–665.
[30] S. Elbaum, A. G. Malishevsky, G. Rothermel, Prioritizing test cases for regression testing, in: Proceedings of the 8th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA'00), 2000, pp. 102–112.
[31] PIT, http://pitest.org/.
[32] PIT operators, http://pitest.org/quickstart/mutators/.
[33] J. H. Andrews, L. C. Briand, Y. Labiche, A. S. Namin, Using mutation analysis for assessing and comparing testing coverage criteria, IEEE Transactions on Software Engineering 32 (8) (2006) 608–624.
[34] D. Schuler, V. Dallmeier, A. Zeller, Efficient mutation testing by checking invariant violations, in: Proceedings of the 18th International Symposium on Software Testing and Analysis (ISSTA'09), 2009, pp. 69–80.
[35] M. Papadakis, Y. Jia, M. Harman, Y. Le Traon, Trivial compiler equivalence: A large scale empirical study of a simple, fast and effective equivalent mutant detection technique, in: Proceedings of the IEEE/ACM 37th IEEE International Conference on Software Engineering (ICSE'15), 2015, pp. 936–946.
[36] R. Just, G. M. Kapfhammer, F. Schweiggert, Do redundant mutants affect the effectiveness and efficiency of mutation analysis?, in: Proceedings of the IEEE 5th International Conference on Software Testing, Verification and Validation (ICST'12), 2012, pp. 720–725.
[37] G. Kaminski, P. Ammann, J. Offutt, Improving logic-based testing, Journal of Systems and Software 86 (8) (2013) 2002–2012.
[38] M. Papadakis, C. Henard, M. Harman, Y. Jia, Y. Le Traon, Threats to the validity of mutation-based test assessment, in: Proceedings of the 25th International Symposium on Software Testing and Analysis (ISSTA'16), 2016, pp. 355–365.
[39] Java: A general-purpose programming language, https://www.oracle.com/java/.
[40] L. Zhang, M. Kim, S. Khurshid, FaultTracer: A change impact and regression fault analysis tool for evolving Java programs, in: Proceedings of the 20th ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE'12), 2012, p. 40.
[41] L. Zhang, M. Kim, S. Khurshid, FaultTracer: A spectrum-based approach to localizing failure-inducing program edits, Journal of Software: Evolution and Process 25 (12) (2013) 1357–1383.
[42] ASM: An all purpose Java bytecode manipulation and analysis framework, http://asm.ow2.org/.
[43] gcc: The GNU Compiler Collection, https://gcc.gnu.org/.
[44] gcov: A Test coverage program, https://gcc.gnu.org/onlinedocs/gcc/Gcov.html.
[45] A. Arcuri, L. Briand, A hitchhiker's guide to statistical tests for assessing randomized algorithms in software engineering, Software Testing, Verification and Reliability 24 (3) (2014) 219–250.
[46] M. Gligoric, A. Groce, C. Zhang, R. Sharma, M. A. Alipour, D. Marinov, Guidelines for coverage-based comparisons of non-adequate test suites, ACM Transactions on Software Engineering and Methodology 24 (4) (2015) 22:1–22:33.
[47] A. Vargha, H. D. Delaney, A critique and improvment of the CL common language effect size statistics of mcgraw and wong, Journal of Education and Behavioral Statistics 25 (2) (2000) 101–132.
[48] R: The R project for statistical computing, https://www.r-project.org/.
[49] Q. Luo, K. Moran, D. Poshyvanyk, M. Di Penta, Assessing test case prioritization on real faults and mutants, in: Proceedings of the 34th IEEE International Conference on Software Maintenance and Evolution (ICSME'18), 2018, pp. 240–251.
[50] B. Miranda, E. Cruciani, R. Verdecchia, A. Bertolino, Fast approaches to scalable similarity-based test case prioritization, in: Proceedings of the 40th International Conference on Software Engineering (ICSE'18), 2018, pp. 222–232.
[51] J. Chen, Y. Bai, D. Hao, L. Zhang, L. Zhang, B. Xie, How do assertions impact coverage-based test-suite reduction?, in: Proceedings of the 10th IEEE International Conference on Software Testing, Verification and Validation (ICST'17), 2017, pp. 418–423.
[52] A. Shi, T. Yung, A. Gyori, D. Marinov, Comparing and combining test-suite reduction and regression test selection, in: Proceedings of the 10th Joint Meeting on Foundations of Software Engineering (ESEC/FSE'15), 2015, pp. 237–247.
[53] L. Zhang, Hybrid regression test selection, in: Proceedings of the 40th International Conference on Software Engineering (ICSE'18), 2018, pp. 199–209.
[54] M. Gligoric, L. Eloussi, D. Marinov, Practical regression test selection with dynamic file dependencies, in: Proceedings of the 24th International Symposium on Software Testing and Analysis (ISSTA'15), 2015, pp. 211–222.
[55] S. Yoo, M. Harman, Regression testing minimization, selection and prioritization: a survey, Software Testing, Verification and Reliability 22 (2) (2012) 67–120.
[56] M. Khatibsyarbini, M. A. Isa, D. N. Jawawi, R. Tumeng, Test case prioritization approaches in regression testing: A systematic literature review, Information and Software Technology 93 (2018) 74–93.
[57] J. A. Jones, M. J. Harrold, Test-suite reduction and prioritization for modified condition/decision coverage, IEEE Transactions on Software Engineering 29 (3) (2003) 195–209.
[58] J. Chi, Y. Qu, Q. Zheng, Z. Yang, W. Jin, D. Cui, T. Liu, Test case prioritization based on method call sequences, in: Proceedings of the 42nd IEEEAnnual Computer Software and Applica-

tions Conference (COMPSAC'18), Vol. 01, 2018, pp. 251–256.
[59] H. Do, G. Rothermel, A. Kinneer, Prioritizing JUnit test cases: An empirical assessment and cost-benefits analysis, Empirical Software Engineering 11 (1) (2006) 33–70.
[60] H. Do, S. Mirarab, L. Tahvildari, G. Rothermel, The effects of time constraints on test case prioritization: A series of controlled experiments, IEEE Transactions on Software Engineering 36 (5) (2010) 593–617.

Rubing Huang received the Ph.D. degree in computer science and technology from the Huazhong University of Science and Technology, Wuhan, China, in 2013. From 2016 to 2018, he was a visiting scholar at Swinburne University of Technology and at Monash University, Australia. He is an associate professor in the Department of Software Engineering, School of Computer Science and Communication Engineering, Jiangsu University, Zhenjiang, China. His current research interests include software testing (including adaptive random testing, random testing, combinatorial testing, and regression testing), debugging, and maintenance. He has more than 50 publications in journals and proceedings, including in IEEE Transactions on Software Engineering, IEEE Transactions on Reliability, Journal of Systems and Software, Information and Software Technology, IET Software, The Computer Journal, International Journal of Software Engineering and Knowledge Engineering, ICSE, ICST, COMPSAC, QRS, SEKE, and SAC. He is a senior member of the China Computer Federation, and a member of the IEEE and the ACM. More about him and his work is available online at https://huangrubing.github.io/.

Quanjun Zhang received the B.Eng. degree in computer science and technology in 2017 from Jiangsu University, Zhenjiang, China, where he is currently working toward the M.Eng. degree with the School of Computer Science and Communication Engineering. His current research interests include software testing and software maintenance.

Dave Towey received the B.A. and M.A. degrees in computer science, linguistics, and languages from the University of Dublin, Trinity College, Ireland; the M.Ed. degree in education leadership from the University of Bristol, U.K.; and the Ph.D. degree in computer science from The University of Hong Kong, China. He is an associate professor at University of Nottingham Ningbo China (UNNC), in Zhejiang, China, where he serves as the director of teaching and learning, and deputy head of school, for the School of Computer Science. He is also the deputy director of the International Doctoral Innovation Centre at UNNC. He is a member of the UNNC Artificial Intelligence and Optimization research group. His current research interests include software testing (especially adaptive random testing, for which he was amongst the earliest researchers who established the field, and metamorphic testing), computer security, and technology-enhanced education. He co-founded the ICSE International Workshop on Metamorphic Testing in 2016. He is a member of both the IEEE and the ACM.

Weifeng Sun received the B.Eng. degree in computer science and technology in 2018 from Jiangsu University, Zhenjiang, China, where he is currently working toward the M.Eng. degree with the School of Computer Science and Communication Engineering. His current research interests include software testing and software debugging. His work has been published in journals and proceedings, including in IEEE Transactions on Software Engineering, IEEE Transactions on Reliability, and the IEEE International Conference on Software Testing, Verification and Validation (ICST). He is a student member of the China Computer Federation and the ACM.

Jinfu Chen received the BE degree in 2004 from Nanchang Hangkong University, Nanchang, China and the PhD degree in 2009 from Huazhong University of Science and Technology, Wuhan, China, both in computer science. He is currently a full professor in the School of Computer Science and Communication Engineering, Jiangsu University, Zhenjiang, China. His major research interests include software testing, software analysis, and trusted software.