Regression Test Case Prioritization by Code Combinations Coverage

Rubing Huang, Quanjun Zhang, Dave Towey, Weifeng Sun, Jinfu Chen
Faculty of Science and Engineering, University of Nottingham Ningbo China, 199 Taikang East Road, Ningbo, 315100, Zhejiang, China.

First published 2020

This work is made available under the terms of the Creative Commons Attribution 4.0 International License:
http://creativecommons.org/licenses/by/4.0

The work is licenced to the University of Nottingham Ningbo China under the Global University Publication Licence:
Regression Test Case Prioritization by Code Combinations Coverage

Rubing Huang\textsuperscript{a}, Quanjun Zhang\textsuperscript{b}, Dave Towey\textsuperscript{c}, Weifeng Sun\textsuperscript{b}, Jinfu Chen\textsuperscript{b}

\textsuperscript{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
\textsuperscript{b}School of Computer Science and Communication Engineering, Jiangsu University, Zhenjiang, Jiangsu 212013, P.R. China
\textsuperscript{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. \cite{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 \cite{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 \cite{7,8,9}.

Traditional RTCP techniques \cite{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 \cite{12}. Because of this, a goal of maximizing code coverage has been incorporated into various RTCP techniques, including greedy strategies \cite{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. \cite{2} proposed two search-based RTCP techniques (a hill-climbing strategy and a genetic strategy) to explore the
search and granularity, and 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 $\lambda$-wise tuples (called $\lambda$-wise parameter-value combinations [16] or $\lambda$-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 $(1, 3, 5), (1, 3, 7), (1, 5, 7), (5, 7)$. Intuitively speaking, when $\lambda = k$, a $\lambda$-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 $\lambda$-wise
parameter-value combinations covered by an abstract test case $tc = (v_1, v_2, \ldots, v_k)$, i.e.,

$$\text{CombSet}(tc, \lambda) = \left\{(v_{i_1}, v_{i_2}, \ldots, v_{i_\lambda}) \mid 1 \leq i_1 < i_2 < \cdots < i_\lambda \leq k\right\}$$

(1)

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

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

(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 $\forall P'' \in PT(P'' \neq P'), f(P') \geq f(P'')$.

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 $U = \{u_1, u_2, \ldots, u_m\}$, and a regression test set $T$ with $n$ test cases ($T = \{tc_1, tc_2, \ldots, tc_n\}$). We define a function $\text{isCovered}(tc, u)$ to measure whether or not a test case $tc$ covers the code unit $u$, as follows:

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

(3)

As a result, each test case $tc$ can be represented by an $m$-wise binary array through the $\text{convertTest}$ function:

$$\text{convertTest}(tc) = (\text{isCovered}(tc, u_1), \text{isCovered}(tc, u_2), \ldots, \text{isCovered}(tc, u_m))$$

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 $u_i$: $\text{isCovered}(tc, u_i)$, where $1 \leq i \leq m$, defined as:

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

(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 \(\lambda\)-wise code combinations coverage (CCC) value of \(tc\) against the test set \(T\) is defined as the number of \(\lambda\)-wise code-unit combinations covered by \(tc\) that are not covered by \(T\):

\[
CCC(tc, T, \lambda) = |\text{CombSet}(\text{convertTest}(tc, \lambda)) \setminus \text{CombSet}(T, \lambda)| 
\]

(5)

where \(T^* = \bigcup_{i \leq \lambda} \text{convertTest}(tc)\).

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 \(\lambda\)-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 will be selected: \(CCC(tc, T, \lambda)\), from Equation (5).

Algorithm 1: \(\text{calculateCombinations}(\text{temp_cover}, \lambda)\)

<table>
<thead>
<tr>
<th>Input:</th>
<th>temp_cover: List of code unit values, (\lambda): Combination strength</th>
</tr>
</thead>
<tbody>
<tr>
<td>Output:</td>
<td>Combinations: Set of (\lambda)-wise code-unit-value combinations</td>
</tr>
</tbody>
</table>

1: Combinations ← \(\emptyset\)  
2: if \(i = 1\) then  
3: for each item \(t\) in temp_cover do  
4: Combinations ← Combinations ∪ \{item\}  
5: end for  
6: return Combinations  
7: end if  
8: num ← |temp_cover|  
9: for each \(i\) \((0 \leq i \leq num - \lambda)\) do  
10: for each item in Combinations \(\{item\} \cup \{item\}\) do  
11: Combinations ← Combinations \(\{item\} \cup \{item\}\)  
12: end for  
13: end for  
14: return Combinations

Algorithm 2: Pseudocode of CCCP

<table>
<thead>
<tr>
<th>Input:</th>
<th>(T: {u_1, u_2, \ldots, u_n}) is a set of unordered test cases with size (n), (\lambda): Combination strength, (U: {a_1, a_2, \ldots, a_m}) is a set of (m) code units for the program (P)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Output:</td>
<td>(S: \text{Prioritized test cases})</td>
</tr>
</tbody>
</table>

1: maximum ← \(-1\)  
2: \(k ← 1\)  
3: for each \(i\) \((1 \leq i \leq n)\) do  
4: Selected\(i\) ← false  
5: num ← 0  
6: for each \(j\) \((1 \leq j \leq m)\) do  
7: \(temp\cdot cover() \leftarrow \text{Cover}(i, u_j)\)  
8: end for  
9: if \(\text{Cover}(i, j)\) \&\& \((j \leq 2)\) 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: \(\text{Cover}(i, j)\) \&\& \(\text{temp\cdot cover()}\)  
18: end if  
19: if not Selected\(i\) then  
20: end if  
21: end for  
22: \(S ← S \cup \{u_k\}\)  
23: end for  
24: flag ← false  
25: while \([|S|] < |\text{CombSet}(T, \lambda)|\) do  
26: if flag then  
27: \(\text{UncoverCombinations}(\text{Combinations}(\lambda))\)  
28: \(S ← S \cup \{u_k\}\)  
29: else  
30: flag ← true  
31: end if  
32: end if  
33: \(\text{Selected}(k)\) ← true  
34: end while  
35: return \(S\)
stores the $\lambda$-wise code-unit-value combinations covered by $tc_i$; and a set $UncoverCombinations$ stores all uncovered $\lambda$-wise code-unit-value combinations. In addition, the variable $flag$ indicates whether or not all $\lambda$-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 $\lambda$-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 $\lambda$-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 $\lambda$-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 $\lambda$-wise code-unit-value combinations against the 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 $S$. In contrast, the total prioritization technique would choose $tc_2$ as the second test case, because $tc_2$ covers more code units than $tc_1$; 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 $S = \{tc_1, tc_3, tc_2\}$.

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 $\lambda$ 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?

---

Fig. 1: An illustrative example of CCCP

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 matrixes based on the function $\text{Cover}(u_i, u_j)$, where $1 \leq i \leq 3$ and $1 \leq j \leq 4$.

<table>
<thead>
<tr>
<th></th>
<th>$u_1$</th>
<th>$u_2$</th>
<th>$u_3$</th>
<th>$u_4$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$t_1$</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>$t_2$</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>$t_3$</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

Consider CCCP with $\lambda = 1$, $UncoverCombinations = \{(1, 2), (3, 4), (5, 6), (7, 8)\}$.

**Step 1**: Randomly choose an element from $\{t_1, t_2\}$ as the first test case in $S$, because each test case covers the same number of units, for example, $S = \{t_2\}$.

**Step 2**: Update $UncoverCombinations = \{(3, 4), (7, 8)\}$, and then calculate

$\text{Candidate}(t_2, 1) = \{(2, 3), (5, 6)\}$;

$\text{Candidate}(t_2, 2) = \{(2, 3), (5, 6)\}$;

$\text{CCCP}(t_2, S, 1) = |UncoverCombinations \cap \text{Candidate}(t_2, 1)| = |\{(2, 3)\}| = 2$;

$\text{CCCP}(t_2, S, 1) = |UncoverCombinations \cap \text{Candidate}(t_2, 1)| = |\{(2, 3)\}| = 2$.

As a result, $t_2$ is selected as the next test case for $S$, i.e., $S = \{t_2\}$.

**Step 3**: Add the last test case $t_2$ for $S$, i.e., $S = \{t_2\}$.
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 $A = 2$ 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.

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:

$$APFD = 1 - \frac{\sum_{i=1}^{m} TF_i}{n \times m} + \frac{1}{2n} \cdot \text{(6)}$$

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:

$$APFD_c = \frac{\sum_{i=1}^{n} (a_i \times (\sum_{j=1}^{n} \beta_j - \frac{1}{m} \alpha_i))}{\sum_{i=1}^{n} \alpha_i \times \sum_{i=1}^{n} \beta_i} \cdot \text{(7)}$$

where $a_1, a_2, \cdots, a_m$ 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:

$$APFD_{c} = \frac{\sum_{i=1}^{n} (\sum_{j=1}^{n} \beta_j - \frac{1}{m} \beta_i)}{m \times \sum_{i=1}^{n} \beta_i} \cdot \text{(8)}$$

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

<table>
<thead>
<tr>
<th>Language</th>
<th>Program</th>
<th>Version</th>
<th>KLC</th>
<th>#Branch</th>
<th>#Method</th>
<th>#Class</th>
<th>#T_Case</th>
<th>#T_Method</th>
<th>#All</th>
<th>#Detected</th>
<th>#SM_Case</th>
<th>#SM_Method</th>
</tr>
</thead>
<tbody>
<tr>
<td>Java</td>
<td>ant</td>
<td>2.42</td>
<td>3.09</td>
<td>3.54</td>
<td>2.34</td>
<td>3.07</td>
<td>3.54</td>
<td>2.95</td>
<td>3.07</td>
<td>3.07</td>
<td>3.07</td>
<td>3.07</td>
</tr>
<tr>
<td></td>
<td>ant    / 2</td>
<td>3.09</td>
<td>3.54</td>
<td>2.34</td>
<td>3.07</td>
<td>3.07</td>
<td>3.54</td>
<td>2.95</td>
<td>3.07</td>
<td>3.07</td>
<td>3.07</td>
<td>3.07</td>
</tr>
<tr>
<td></td>
<td>ant    / 3</td>
<td>3.09</td>
<td>3.54</td>
<td>2.34</td>
<td>3.07</td>
<td>3.07</td>
<td>3.54</td>
<td>2.95</td>
<td>3.07</td>
<td>3.07</td>
<td>3.07</td>
<td>3.07</td>
</tr>
<tr>
<td></td>
<td>jmeter</td>
<td>4.2</td>
<td>3.54</td>
<td>2.34</td>
<td>3.07</td>
<td>3.07</td>
<td>3.54</td>
<td>2.95</td>
<td>3.07</td>
<td>3.07</td>
<td>3.07</td>
<td>3.07</td>
</tr>
<tr>
<td></td>
<td>jmeter  / 2</td>
<td>3.54</td>
<td>2.34</td>
<td>3.07</td>
<td>3.07</td>
<td>3.54</td>
<td>2.95</td>
<td>3.07</td>
<td>3.07</td>
<td>3.07</td>
<td>3.07</td>
<td>3.07</td>
</tr>
<tr>
<td></td>
<td>jmeter  / 3</td>
<td>3.54</td>
<td>2.34</td>
<td>3.07</td>
<td>3.07</td>
<td>3.54</td>
<td>2.95</td>
<td>3.07</td>
<td>3.07</td>
<td>3.07</td>
<td>3.07</td>
<td>3.07</td>
</tr>
<tr>
<td></td>
<td>jtopas</td>
<td>3.09</td>
<td>3.54</td>
<td>2.34</td>
<td>3.07</td>
<td>3.07</td>
<td>3.54</td>
<td>2.95</td>
<td>3.07</td>
<td>3.07</td>
<td>3.07</td>
<td>3.07</td>
</tr>
<tr>
<td></td>
<td>xmlsec</td>
<td>3.09</td>
<td>3.54</td>
<td>2.34</td>
<td>3.07</td>
<td>3.07</td>
<td>3.54</td>
<td>2.95</td>
<td>3.07</td>
<td>3.07</td>
<td>3.07</td>
<td>3.07</td>
</tr>
</tbody>
</table>

Table 2: Subject program details
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 subsuming 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 remove 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: \( P_{00}, P_{V1}, P_{V2}, P_{V3}, P_{V4}, \) 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\(_{exc}\) and \( M \), where \( M \in \{TCP\_utt, TCP\_add, TCP\_utt, TCP\_search\} \), the symbol \( \checkmark \) means that TCP\(_{exc}\) is better (\( p\)-value is less than 0.05, and the effect size \( \hat{A}_{12}(TCP\_exc, M) \) is greater than 0.50); the symbol \( \bullet \) means that \( M \) is better (the \( p\)-value is less than 0.05, and \( \hat{A}_{12}(TCP\_exc, M) \) is less than 0.50); and the symbol \( \circ \) 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.
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: The effectiveness comparison of APFD for Java programs at the class-test level. For a comparison between two methods TCP$_{fee}$ and $M$, where $M \in \{TCP_{hot}, TCP_{add}, TCP_{add}, TCP_{search}\}$, the symbol $\vartriangleright$ means that TCP$_{fee}$ is better (p-value is less than 0.05, and the effect size $\hat{A}_{12}(TCP_{fee}, M)$ is greater than 0.50); the symbol $\vartriangleleft$ means that $M$ is better (the $p$-value is less than 0.05, and $\hat{A}_{12}(TCP_{fee}, M)$ is less than 0.50); and the symbol $\equiv$ means that there is no statistically significant difference between them (the $p$-value is greater than 0.05).

### Table 3: Statistical effectiveness comparisons of APFD for Java programs at the class-test level.

<table>
<thead>
<tr>
<th>Program Name</th>
<th>Statement Coverage</th>
<th>Method Coverage</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>TCP$_{fee}$</td>
<td>TCP$_{hot}$</td>
</tr>
<tr>
<td>antri</td>
<td>$\vartriangleright$</td>
<td>$\vartriangleleft$</td>
</tr>
<tr>
<td>ant</td>
<td>$\vartriangleleft$</td>
<td>$\equiv$</td>
</tr>
<tr>
<td>antbr</td>
<td>$\equiv$</td>
<td>$\equiv$</td>
</tr>
<tr>
<td>ant</td>
<td>$\equiv$</td>
<td>$\equiv$</td>
</tr>
<tr>
<td>antbr</td>
<td>$\equiv$</td>
<td>$\equiv$</td>
</tr>
<tr>
<td>antbr</td>
<td>$\equiv$</td>
<td>$\equiv$</td>
</tr>
</tbody>
</table>

Table 4: The effectiveness comparison of APFD for Java programs at the test-method level. For a comparison between two methods TCP$_{fee}$ and $M$, where $M \in \{TCP_{hot}, TCP_{add}, TCP_{add}, TCP_{search}\}$, the symbol $\vartriangleright$ means that TCP$_{fee}$ is better (p-value is less than 0.05, and the effect size $\hat{A}_{12}(TCP_{fee}, M)$ is greater than 0.50); the symbol $\vartriangleleft$ means that $M$ is better (the $p$-value is less than 0.05, and $\hat{A}_{12}(TCP_{fee}, M)$ is less than 0.50); and the symbol $\equiv$ means that there is no statistically significant difference between them (the $p$-value is greater than 0.05).

### Table 4: Statistical effectiveness comparisons of APFD for Java programs at the test-method level.

<table>
<thead>
<tr>
<th>Program Name</th>
<th>Statement Coverage</th>
<th>Method Coverage</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>TCP$_{fee}$</td>
<td>TCP$_{hot}$</td>
</tr>
<tr>
<td>antri</td>
<td>$\vartriangleright$</td>
<td>$\vartriangleleft$</td>
</tr>
<tr>
<td>ant</td>
<td>$\vartriangleleft$</td>
<td>$\equiv$</td>
</tr>
<tr>
<td>antbr</td>
<td>$\equiv$</td>
<td>$\equiv$</td>
</tr>
<tr>
<td>ant</td>
<td>$\equiv$</td>
<td>$\equiv$</td>
</tr>
<tr>
<td>antbr</td>
<td>$\equiv$</td>
<td>$\equiv$</td>
</tr>
<tr>
<td>antbr</td>
<td>$\equiv$</td>
<td>$\equiv$</td>
</tr>
<tr>
<td>antbr</td>
<td>$\equiv$</td>
<td>$\equiv$</td>
</tr>
</tbody>
</table>

(4) There is a statistically significant difference between TCP$_{fee}$ and TCP$_{att}$, which supports the above observations. However, none of the other three techniques (TCP$_{hot}$, TCP$_{add}$, or TCP$_{search}$) is either always better or always worse than TCP$_{fee}$, with TCP$_{fee}$ 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$_{fee}$ performs better than TCP$_{att}$ and TCP$_{add}$. 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$_{att}$ 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
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 TCPart.

(4) Other than for a few cases (such as jtopas_v2 with method coverage, and xmlsec_v1 with statement coverage), CCCP usually has better performance than TCPsearch.

(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 add, 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 comparison-
s 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
(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_{art} 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_m, and M, where M ∈ {TCP_m, TCP_add, TCP_unt, TCP_search}, the symbol $\triangledown$ means that TCP_m is better (p-value is less than 0.05, and the effect size $\Delta_{A}$TCP_m,M) is greater than 0.50; the symbol $\triangledown$ means that M is better (the p-value is less than 0.05, and $\Delta_{A}$TCP_m,M is less than 0.50); and the symbol $\triangledown$ means that there is no statistically significant difference between them (the p-value is greater than 0.05).

<table>
<thead>
<tr>
<th>Program Name</th>
<th>TCP_m</th>
<th>TCP_add</th>
<th>TCP_unt</th>
<th>TCP_search</th>
<th>Branch Coverage</th>
<th>Method Coverage</th>
</tr>
</thead>
<tbody>
<tr>
<td>flex_v1</td>
<td>0.91</td>
<td>0.91</td>
<td>0.91</td>
<td>0.91</td>
<td>0.91</td>
<td>0.91</td>
</tr>
<tr>
<td>flex_v2</td>
<td>0.92</td>
<td>0.91</td>
<td>0.91</td>
<td>0.91</td>
<td>0.91</td>
<td>0.91</td>
</tr>
<tr>
<td>flex_v3</td>
<td>0.93</td>
<td>0.91</td>
<td>0.91</td>
<td>0.91</td>
<td>0.91</td>
<td>0.91</td>
</tr>
<tr>
<td>flex_v4</td>
<td>0.94</td>
<td>0.91</td>
<td>0.91</td>
<td>0.91</td>
<td>0.91</td>
<td>0.91</td>
</tr>
<tr>
<td>flex_v5</td>
<td>0.95</td>
<td>0.91</td>
<td>0.91</td>
<td>0.91</td>
<td>0.91</td>
<td>0.91</td>
</tr>
<tr>
<td>grep_v1</td>
<td>0.96</td>
<td>0.91</td>
<td>0.91</td>
<td>0.91</td>
<td>0.91</td>
<td>0.91</td>
</tr>
<tr>
<td>grep_v2</td>
<td>0.97</td>
<td>0.91</td>
<td>0.91</td>
<td>0.91</td>
<td>0.91</td>
<td>0.91</td>
</tr>
<tr>
<td>grep_v3</td>
<td>0.98</td>
<td>0.91</td>
<td>0.91</td>
<td>0.91</td>
<td>0.91</td>
<td>0.91</td>
</tr>
<tr>
<td>grep_v4</td>
<td>0.99</td>
<td>0.91</td>
<td>0.91</td>
<td>0.91</td>
<td>0.91</td>
<td>0.91</td>
</tr>
<tr>
<td>gzip_v1</td>
<td>1.00</td>
<td>0.91</td>
<td>0.91</td>
<td>0.91</td>
<td>0.91</td>
<td>0.91</td>
</tr>
<tr>
<td>tcp_w</td>
<td>0.94</td>
<td>0.91</td>
<td>0.91</td>
<td>0.91</td>
<td>0.91</td>
<td>0.91</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Language</th>
<th>Status</th>
<th>Statement Coverage</th>
<th>Branch Coverage</th>
<th>Method Coverage</th>
<th>Sum $\Sigma$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Java (test-class)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>5</td>
<td>0</td>
<td>13</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>6</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>7</td>
<td>13</td>
<td>1</td>
<td>11</td>
<td>11</td>
</tr>
<tr>
<td>Java (test-method)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>14</td>
<td>6</td>
<td>11</td>
<td>10</td>
<td>14</td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>1</td>
<td>3</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>7</td>
<td>0</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>C</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>0</td>
<td>9</td>
<td>9</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>0</td>
<td>12</td>
<td>12</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>20</td>
<td>12</td>
<td>15</td>
<td>4</td>
</tr>
</tbody>
</table>

Table 6 summarizes the statistical results, presenting the total number of times TCP_m is better ($\triangledown$), worse ($\triangledown$), or not statistically different ($\triangledown$), 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_m performs much better than TCP_add, similarly to TCP_un and TCP_search, and slightly worse than TCP_search.
2. When prioritizing Java test suites at the test-method level, TCP_m performs much better than TCP_add.
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 C-CCP 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<sub>c</sub> for Java programs at the test-class level. For a comparison between two methods TCP<sub>art</sub> and M, where M ∈ {TCP<sub>art</sub>, TCP<sub>old</sub>, TCP<sub>prot</sub>, TCP<sub>catch</sub>}, the symbol < means that TCP<sub>art</sub> is better (p-value is less than 0.05, and the effect size Λ<sub>c</sub>(TCP<sub>art</sub>, M) ≥ 0.50); the symbol ≥ means that M is better (the p-value is less than 0.05, and Λ<sub>c</sub>(TCP<sub>art</sub>, M) < 0.50); and the symbol = means that there is no statistically significant difference between them (the p-value is greater than 0.05).

<table>
<thead>
<tr>
<th>Program Name</th>
<th>TCP&lt;sub&gt;art&lt;/sub&gt;</th>
<th>TCP&lt;sub&gt;old&lt;/sub&gt;</th>
<th>TCP&lt;sub&gt;prot&lt;/sub&gt;</th>
<th>TCP&lt;sub&gt;catch&lt;/sub&gt;</th>
<th>Method Coverage</th>
</tr>
</thead>
<tbody>
<tr>
<td>ant&lt;sub&gt;v1&lt;/sub&gt;</td>
<td>(0.10)</td>
<td>(1.00)</td>
<td>(0.97)</td>
<td>(1.00)</td>
<td>(0.58)</td>
</tr>
<tr>
<td>ant&lt;sub&gt;v2&lt;/sub&gt;</td>
<td>(0.10)</td>
<td>(0.75)</td>
<td>(0.59)</td>
<td>(0.70)</td>
<td>(1.00)</td>
</tr>
<tr>
<td>ant&lt;sub&gt;v3&lt;/sub&gt;</td>
<td>(0.10)</td>
<td>(0.65)</td>
<td>(0.98)</td>
<td>(0.66)</td>
<td>(1.00)</td>
</tr>
<tr>
<td>ant&lt;sub&gt;v4&lt;/sub&gt;</td>
<td>(0.10)</td>
<td>(0.60)</td>
<td>(0.85)</td>
<td>(0.59)</td>
<td>(0.83)</td>
</tr>
<tr>
<td>jmeter&lt;sub&gt;v1&lt;/sub&gt;</td>
<td>(0.00)</td>
<td>(0.00)</td>
<td>(1.00)</td>
<td>(0.00)</td>
<td>(0.00)</td>
</tr>
<tr>
<td>jmeter&lt;sub&gt;v2&lt;/sub&gt;</td>
<td>(0.00)</td>
<td>(0.14)</td>
<td>(0.99)</td>
<td>(0.12)</td>
<td>(0.00)</td>
</tr>
<tr>
<td>jmeter&lt;sub&gt;v3&lt;/sub&gt;</td>
<td>(0.00)</td>
<td>(0.66)</td>
<td>(0.69)</td>
<td>(0.67)</td>
<td>(0.00)</td>
</tr>
<tr>
<td>jmeter&lt;sub&gt;v4&lt;/sub&gt;</td>
<td>(0.49)</td>
<td>(0.73)</td>
<td>(0.78)</td>
<td>(0.75)</td>
<td>(0.00)</td>
</tr>
<tr>
<td>jmeter&lt;sub&gt;v5&lt;/sub&gt;</td>
<td>(0.10)</td>
<td>(0.61)</td>
<td>(0.92)</td>
<td>(0.64)</td>
<td>(0.92)</td>
</tr>
<tr>
<td>jmeter&lt;sub&gt;v6&lt;/sub&gt;</td>
<td>(0.33)</td>
<td>(0.49)</td>
<td>(0.59)</td>
<td>(0.30)</td>
<td>(0.18)</td>
</tr>
<tr>
<td>jmeter&lt;sub&gt;v7&lt;/sub&gt;</td>
<td>(0.10)</td>
<td>(0.10)</td>
<td>(0.69)</td>
<td>(1.00)</td>
<td>(1.00)</td>
</tr>
<tr>
<td>jmeter&lt;sub&gt;v8&lt;/sub&gt;</td>
<td>(0.10)</td>
<td>(0.00)</td>
<td>(0.49)</td>
<td>(1.00)</td>
<td>(1.00)</td>
</tr>
<tr>
<td>jmeter&lt;sub&gt;v9&lt;/sub&gt;</td>
<td>(0.10)</td>
<td>(0.50)</td>
<td>(0.12)</td>
<td>(0.50)</td>
<td>(1.00)</td>
</tr>
<tr>
<td>jmeter&lt;sub&gt;v10&lt;/sub&gt;</td>
<td>(0.78)</td>
<td>(0.61)</td>
<td>(0.50)</td>
<td>(0.61)</td>
<td>(0.01)</td>
</tr>
<tr>
<td>jmeter&lt;sub&gt;v11&lt;/sub&gt;</td>
<td>(0.00)</td>
<td>(0.50)</td>
<td>(0.87)</td>
<td>(0.50)</td>
<td>(0.00)</td>
</tr>
<tr>
<td>jmeter&lt;sub&gt;v12&lt;/sub&gt;</td>
<td>(0.00)</td>
<td>(0.50)</td>
<td>(0.22)</td>
<td>(0.50)</td>
<td>(1.00)</td>
</tr>
<tr>
<td>jmeter&lt;sub&gt;v13&lt;/sub&gt;</td>
<td>(0.00)</td>
<td>(0.08)</td>
<td>(0.27)</td>
<td>(0.00)</td>
<td>(0.00)</td>
</tr>
<tr>
<td>jmeter&lt;sub&gt;v14&lt;/sub&gt;</td>
<td>(0.52)</td>
<td>(0.34)</td>
<td>(0.38)</td>
<td>(0.44)</td>
<td>(0.28)</td>
</tr>
<tr>
<td>jmeter&lt;sub&gt;v15&lt;/sub&gt;</td>
<td>(0.52)</td>
<td>(0.33)</td>
<td>(0.34)</td>
<td>(0.34)</td>
<td>(0.28)</td>
</tr>
</tbody>
</table>

All Java Programs: (0.52) (0.32) (0.34) (0.34) (0.16) (0.47) (0.54) (0.47) (0.47) (0.40) (0.67) (0.54)

5.2.1. APFD<sub>c</sub> Results: Java Programs (Test-Class Level)

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

1. Compared with TCP<sub>art</sub>, TCP<sub>prot</sub> has much better APFD<sub>c</sub> 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<sub>prot</sub> performs worse than TCP<sub>art</sub>, with a maximum APFD<sub>c</sub> difference of about 15%.

2. Although TCP<sub>ecc</sub> performs better than TCP<sub>art</sub> in many cases (for example, with jmeter, for all code coverage granularities), it also sometimes performs worse (including with ant<sub>v2</sub> and ant<sub>v3</sub> for the branch and method coverage levels). Nevertheless, the TCP<sub>ecc</sub> APFD<sub>c</sub> values have much lower variation than TCP<sub>art</sub>.

3. TCP<sub>ecc</sub> has very similar performance to TCP<sub>old</sub> and TCP<sub>catch</sub>, sometimes performing slightly better (for example, with jtopas, using statement coverage) or worse (such as with xmlsec<sub>v3</sub>, for method coverage). The differences among the mean and median APFD<sub>c</sub> values for the three RTCP techniques are very small, at
(4) Overall, the statistical analyses support the box plot observations. Looking at all Java programs together, \( TCP_{cc} \) 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, \( TCP_{cc} \) performs similarly, or slightly better, compared with \( TCP_{add} \) and \( TCP_{reach} \), with the \( \hat{A}_{12} \) values ranging from 0.50 to 0.53.

5.2.2. APFD, 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, \( ant\_v1 \)), \( TCP_{cc} \) has much better APFD\(_c\) performance than \( TCP_{tot} \), with the maximum mean and median differences reaching about 50%.

(2) \( TCP_{cc} \) 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 \( TCP_{cc} \) for the program \( jtopas \). For the program \( xmlsec \), however, neither \( TCP_{art} \) nor \( TCP_{cc} \) is always better: At branch coverage level, for example, \( TCP_{cc} \) is better for version \( v2 \); \( TCP_{art} \) is better for version \( v3 \); and they have similar performance for version \( v1 \).

(3) The \( TCP_{cc} \) 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<sub>i</sub> for C programs. For a comparison between two methods TCP<sub>art</sub> and M, where M ∈ {TCP<sub>art</sub>, TCP<sub>add</sub>, TCP<sub>math</sub>, TCP<sub>stat</sub>}; the symbol √ means that TCP<sub>art</sub> is better (p-value is less than 0.05, and the effect size Λ<sub>12</sub>(TCP<sub>art</sub>, M) is greater than 0.50); the symbol ♦ means that M is better (the p-value is less than 0.05, and Λ<sub>12</sub>(TCP<sub>art</sub>, 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).

<table>
<thead>
<tr>
<th>Program Name</th>
<th>Statement Coverage</th>
<th>Branch Coverage</th>
<th>Method Coverage</th>
</tr>
</thead>
<tbody>
<tr>
<td>gcc</td>
<td>(0.03)</td>
<td>0.67</td>
<td>0.68</td>
</tr>
<tr>
<td>g9ip&lt;sub&gt;3&lt;/sub&gt;</td>
<td>(0.01)</td>
<td>0.61</td>
<td>0.61</td>
</tr>
<tr>
<td>g9ip&lt;sub&gt;4&lt;/sub&gt;</td>
<td>(0.01)</td>
<td>0.60</td>
<td>0.60</td>
</tr>
<tr>
<td>g9ip&lt;sub&gt;5&lt;/sub&gt;</td>
<td>(0.01)</td>
<td>0.60</td>
<td>0.60</td>
</tr>
<tr>
<td>make</td>
<td>(0.03)</td>
<td>0.64</td>
<td>0.64</td>
</tr>
<tr>
<td>make&lt;sub&gt;2&lt;/sub&gt;</td>
<td>(0.01)</td>
<td>0.61</td>
<td>0.61</td>
</tr>
<tr>
<td>make&lt;sub&gt;3&lt;/sub&gt;</td>
<td>(0.01)</td>
<td>0.60</td>
<td>0.60</td>
</tr>
<tr>
<td>make&lt;sub&gt;4&lt;/sub&gt;</td>
<td>(0.04)</td>
<td>0.59</td>
<td>0.59</td>
</tr>
<tr>
<td>make&lt;sub&gt;5&lt;/sub&gt;</td>
<td>(0.01)</td>
<td>0.60</td>
<td>0.60</td>
</tr>
<tr>
<td>sed</td>
<td>(0.01)</td>
<td>0.56</td>
<td>0.56</td>
</tr>
<tr>
<td>All C Programs</td>
<td>(0.03)</td>
<td>0.67</td>
<td>0.67</td>
</tr>
</tbody>
</table>

(continued)

(4) Although there are some large performance differences between TCP<sub>art</sub> and TCP<sub>math</sub> (such as for ant<sub>y</sub>2 and xsmssec, with branch coverage), overall, they have similar mean and median APFD<sub>i</sub> values. In most cases, TCP<sub>art</sub> has lower variation in APFD<sub>i</sub> values than TCP<sub>math</sub>.

(5) Overall, the statistical analyses support the box plot observations. Looking at all Java programs together, TCP<sub>art</sub> is better than TCP<sub>math</sub> and TCP<sub>stat</sub>, with p-values much less than 0.05, and the Λ<sub>12</sub> values ranging from 0.62 to 0.87. TCP<sub>math</sub> is better than TCP<sub>stat</sub>, with the p-values less than 0.05, and the Λ<sub>12</sub> values ranging from 0.51 to 0.57. Finally, TCP<sub>art</sub> and TCP<sub>add</sub> have very similar performance: The Λ<sub>12</sub> 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<sub>i</sub> Results: C Programs

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

1. Except for some very few cases (such as g9ip<sub>3</sub>, g9ip<sub>4</sub>, and g9ip<sub>5</sub>, with method coverage), TCP<sub>art</sub> has much higher APFD<sub>i</sub> values than TCP<sub>math</sub> and TCP<sub>add</sub>, with the maximum mean and median APFD<sub>i</sub> differences between TCP<sub>art</sub> and TCP<sub>math</sub> reaching more than 35%; and between TCP<sub>art</sub> and TCP<sub>add</sub> being about 15%.

2. TCP<sub>art</sub> performs differently compared with TCP<sub>math</sub> and TCP<sub>stat</sub> for different programs and different code coverage granularities: With the program flex, for example, for all versions and code coverage granularities, TCP<sub>art</sub> is more effective; however, with make, for both statement and branch coverage, TCP<sub>art</sub> and TCP<sub>stat</sub> 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<sub>art</sub> and TCP<sub>stat</sub>, TCP<sub>add</sub>, TCP<sub>art</sub> and TCP<sub>math</sub>, and TCP<sub>art</sub> and TCP<sub>math</sub> are less than 0.05. indicating that the APFD<sub>i</sub> scores are all significantly different. According to the effect size Λ<sub>12</sub> values, TCP<sub>art</sub> outperforms TCP<sub>stat</sub> and TCP<sub>add</sub> and performs slightly better than TCP<sub>math</sub> and TCP<sub>art</sub> (except at the method coverage level).

Table 10 summarizes the statistical results, presenting the total number of times TCP<sub>art</sub> is better (√), worse (♦), or not statistically different (⊤), compared to the other RTK techniques. Based on this table, we can
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$_{add}$, and TCP$_{search}$; and slightly better than TCP$_{art}$.

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
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{est, perf} performing better (✓) rather than worse (✘) than TCP{est, add, art}, TCP{art, est} and TCP{search, est} 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{est}.

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{est} results (Figures 5 to 7), in spite of some cases where the three types of code coverage provide very different APFD or APFD{est} 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{est} 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 y-axis; and the APFD{est} 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{est} values.

Table 11 presents a comparison of the mean and median APFD and APFD{est} values, and also shows the p-values/effect size A_{12} for the different code coverage granularity comparisons. It can be seen from the table that the APFD and APFD{est} 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{est} 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{est} 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

![Fig. 8: Effectiveness: CCCP APFD and APFD_{est} results with different code coverage and test case granularities for all programs](image-url)
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 \( \mu/\sigma \) (where \( \mu \) is the mean time and \( \sigma \) 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 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_add, and very similar time to TCP_add. irrespective of subject program, and code coverage and test case granularities. Also, because TCP_art 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_add, has similar speed to TCP_add, but performs slower than TCP_art.

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_art and TCP_add denote CCCP with \( \lambda = 1 \) and \( \lambda = 2 \), respectively. Table 13 presents the statistical comparisons of the TCP_ccc APFD and APFDC scores with those of the other five RTCP techniques: Each data cell shows the \( p \)-value/efffect size \( \hat{\lambda}_{12} \) value.

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

(1) TCP_ccc2 has the higher mean and median APFD and APFDC values than TCP_art and TCP_add, and better or similar to TCP_art, TCP_add, 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 11: Statistical effectiveness comparisons of APFD and APFDC between different granularities for CCCP. Each cell in the Mean, Median, and Comparison columns represents the mean APFD or APFDC, value, the median value, and the \( p \)-values/efffect size \( \hat{\lambda}_{12} \) for the different code coverage granularity comparisons, respectively.

<table>
<thead>
<tr>
<th>Metric</th>
<th>Language</th>
<th>Mean</th>
<th>Median</th>
<th>Comparison</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Statement</td>
<td>Branch</td>
<td>Method</td>
</tr>
<tr>
<td>APFD</td>
<td>Java (test-class)</td>
<td>0.5385</td>
<td>0.5440</td>
<td>0.5369</td>
</tr>
<tr>
<td></td>
<td>Java (test-method)</td>
<td>0.6542</td>
<td>0.6646</td>
<td>0.6668</td>
</tr>
<tr>
<td></td>
<td>C</td>
<td>0.6115</td>
<td>0.6156</td>
<td>0.6180</td>
</tr>
<tr>
<td>APFDC</td>
<td>Java (test-class)</td>
<td>0.6270</td>
<td>0.6270</td>
<td>0.6270</td>
</tr>
<tr>
<td></td>
<td>Java (test-method)</td>
<td>0.7130</td>
<td>0.6972</td>
<td>0.7227</td>
</tr>
<tr>
<td></td>
<td>C</td>
<td>0.7222</td>
<td>0.7358</td>
<td>0.7581</td>
</tr>
</tbody>
</table>

20
<table>
<thead>
<tr>
<th>Language</th>
<th>Program</th>
<th>Time</th>
<th>Branch Coverage</th>
<th>Statement Coverage</th>
<th>Method Coverage</th>
<th>Sum ( \sum )</th>
</tr>
</thead>
<tbody>
<tr>
<td>[TCP]</td>
<td>ccc</td>
<td>1432</td>
<td>39.8</td>
<td>3.8</td>
<td>0.1</td>
<td>0.5</td>
</tr>
<tr>
<td></td>
<td>jtopas</td>
<td>1339</td>
<td>41.5</td>
<td>3.8</td>
<td>0.1</td>
<td>0.5</td>
</tr>
<tr>
<td></td>
<td>Java</td>
<td>1310</td>
<td>41.35</td>
<td>3.8</td>
<td>0.1</td>
<td>0.5</td>
</tr>
<tr>
<td></td>
<td>jmeter</td>
<td>1327</td>
<td>22.31</td>
<td>3.8</td>
<td>0.1</td>
<td>0.5</td>
</tr>
<tr>
<td></td>
<td>xmlsec</td>
<td>1335</td>
<td>20.25</td>
<td>3.8</td>
<td>0.1</td>
<td>0.5</td>
</tr>
<tr>
<td></td>
<td>Sum</td>
<td>1281</td>
<td>21.78</td>
<td>3.8</td>
<td>0.1</td>
<td>0.5</td>
</tr>
</tbody>
</table>
TCP_{cc2} and each of other five RTCP techniques, regardless of APFD and APFD_ci values. The $\lambda_1$ results also show TCP_{cc2} to outperform TCP_{add}, TCP_{art}, TCP_{search}, and TCP_{cc1}, 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 $\lambda$ 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_{cc2} 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_{cc2} 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_{cc2} 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_{cc2} 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_{cc2} 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).

### 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/huangerubing/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 APFDc 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 APFDc 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 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-
duction coverage. Based on this, we proposed a new prioritization technique, code combinations coverage based priority (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. KY-CX19_1614.

References


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.