A Unified Test Case Prioritization Approach

Dan Hao, Peking University
Lingming Zhang, University of Texas at Dallas
Lu Zhang, Peking University
Gregg Rothermel, University of Nebraska
Hong Mei, Peking University

Test case prioritization techniques attempt to re-order test cases in a manner that increases the rate at which faults are detected during regression testing. Coverage-based test case prioritization techniques typically use one of two overall strategies, a total strategy or an additional strategy. These strategies prioritize test cases based on the total number of code (or code-related) elements covered per test case and the number of additional (not-yet-covered) code (or code-related) elements covered per test case, respectively. In this article, we present a unified test case prioritization approach that encompasses both the total and additional strategies. Our unified test case prioritization approach includes two models ("basic" and "extended") by which a spectrum of test case prioritization techniques ranging from a purely total to a purely additional technique can be defined by specifying the value of a parameter referred to as the $f_p$ value. To evaluate our approach, we performed an empirical study on 28 Java objects and 40 C objects, considering the impact of three internal factors (model type, choice of $f_p$ value, and coverage type) and three external factors (coverage granularity, test case granularity, and programming/testing paradigm), all of which can be manipulated by our approach. Our results demonstrate that a wide range of techniques derived from our basic and extended models with uniform $f_p$ values can outperform purely total techniques, and are competitive with purely additional techniques. Considering the influence of each internal and external factor studied, the results demonstrate that various values of each factor have non-trivial influence on test case prioritization techniques.

Categories and Subject Descriptors: D.2.5 [Software Engineering]: Testing and Debugging

General Terms: Experimentation, Verification

Additional Key Words and Phrases: Software testing, test case prioritization, total strategy, additional strategy

ACM Reference Format:

1. INTRODUCTION

When software engineers release new versions of software systems, they retest them, and this process is known as “regression testing”. One approach to regression testing involves re-using existing test suites, but in practice this can be expensive. For example, Elbaum et al. [Elbaum et al. 2000; Elbaum et al. 2002]
report one industrial system for which the time required to execute an entire regression test suite was seven weeks. Test case prioritization techniques (e.g., [Elbaum et al. 2000; Elbaum et al. 2001; Elbaum et al. 2002; Qu et al. 2008; Rothermel et al. 1999; Wong et al. 1997]) attempt to address this problem, by reordering test cases such that they meet testing goals earlier. One such goal involves detection of faults, and if test cases are prioritized to detect faults faster, this can reduce the costs associated with terminating testing processes early, and help engineers begin debugging tasks early [Xie et al. 2014].

Most existing research on test case prioritization focuses on prioritization techniques themselves. The vast majority of prioritization techniques defined to date (Section 6 provides details) rely on one of two overall strategies, usually described as the total and additional strategies. Given a coverage criterion, a total strategy sorts test cases in descending order of the number of code elements that they cover, whereas an additional strategy iteratively selects a next test case that yields the maximal coverage of code elements not yet covered by previously prioritized test cases. Empirical results have shown [Mei et al. 2012] that the additional strategy is usually more effective than the total strategy in terms of its ability to improve a test suite’s rate of fault detection; thus, this strategy is often considered to be the most useful baseline test case prioritization strategy [Jiang et al. 2009; Mei et al. 2012].

Despite prior empirical results, the total and additional strategies do have different strengths and weaknesses. Suppose that statement coverage is being used to test program $P$. With the total strategy, after a test case $t$ is chosen for $P$, all statements covered by $t$ are considered again when choosing a next test case. If $P$ contains only one fault $f$ in the statements covered by $t$, and if $t$ detects $f$, it is not necessary to consider the statements covered by $t$ again when choosing a next test case. The additional strategy avoids this problem by focusing on statements not yet covered by $t$ when choosing a next test case. Conversely, however, if $t$ does not detect $f$, but $f$ is detectable by other test cases, detection of $f$ by the additional strategy may be delayed while that strategy focuses on statements that have not yet been covered, whereas the total strategy may detect $f$ earlier. More generally, the total strategy may delay the detection of faults in statements that are covered infrequently, whereas the additional strategy may delay the detection of faults in statements that are covered frequently (though these effects also vary with the probability that a test case that covers a faulty statement will also expose the fault).

We conjecture that a prioritization approach that combines aspects of the additional and total strategies may be more effective than either of these strategies alone. In this paper, we define a unified test case prioritization approach that facilitates such a combination. In particular, we define two prioritization models (a “basic” model and an “extended” model) that allow us to derive a spectrum of prioritization techniques. The spectrum is achieved by utilizing estimates of the likelihood that a test case may detect faults in a covered program element, a likelihood that is encapsulated in a parameter that we refer to as the “$f_p$ value”. The total ($f_p = 0$) and additional ($f_p = 1$) strategies account for two extreme instances of our prioritization approach.

Because the fault-detection capabilities of test cases may differ for various reasons (e.g., differences in the structure of the source code under test), we further extend our prioritization models by using differentiated $f_p$ values. Furthermore, because it can be difficult, in practice, to collect dynamic coverage information for test cases, we extend our unified test case prioritization approach to use either dynamic or static coverage information. Given these extensions, ultimately, our unified test case prioritization approach yields a family of different prioritization techniques that differ based on the values of the three internal factors: model type, $f_p$ value, and coverage type.
We performed an empirical study of a large number of programs (including 40 C and 28 Java programs), investigating the effectiveness of our unified approach, considering the impact of the three internal factors just described. At the same time, we also considered three external factors that can potentially affect prioritization effectiveness: coverage granularity (the level of code at which coverage is measured), test case granularity (the level of program unit on which test cases are executed), and programming/testing paradigm. Our study shows that in general, based on various values of the three internal factors, our unified approach can generate a spectrum of test case prioritization techniques, most of which are more effective than the total strategy and as effective as the additional strategy. Furthermore, some test case prioritization techniques (e.g., the unified approach with differentiated \( f_p \) values) are more effective than the additional strategy. Comparing strategies using different values of factors, we also find that these factors can affect prioritization results. In particular, with respect to results measured in our specific experiment on our particular objects of study:

- The unified approach using the basic model is typically less effective than the unified approach using the extended model.
- The unified approach using uniform \( f_p \) values is typically less effective than the unified approach using differentiated \( f_p \) values.
- Although the unified approach based on static coverage information is typically less effective than the unified approach based on dynamic coverage information, there are cases in which these two approaches have no significant differences.
- The unified approach using statement or method-level coverage differs only trivially; thus, using method-level coverage may be more cost-effective given the reduced costs of code instrumentation associated with it.
- The unified approach is significantly more effective when it is used to prioritize test cases that operate at the test-method level than when used to prioritize test cases that operate at the test-class level.
- The unified approach is more effective when it is used to prioritize test cases for Java programs with unit tests than for C programs with system tests.

The main contributions of this paper are as follows.

- A novel unified test case prioritization approach that combines the total and additional strategies and yields a spectrum of prioritization techniques having advantages of both the total and additional strategies.
- Empirical evidence that many prioritization techniques derived from our approach and lying on the spectrum in between purely total and additional techniques are more effective than the former techniques and at least competitive with the latter techniques.
- An empirical evaluation of the influence of three internal factors and three external factors on the effectiveness of the prioritization techniques created through our unified approach.

The remainder of this article is organized as follows. Section 2 presents our approach. Section 3 presents the design and operational details of our empirical study. Section 4 presents the main findings of our study. Section 5 discusses deeper issues involved in the proposed approach. Section 6 discusses related research. Finally, Section 7 concludes and discusses potential future work.
2. A UNIFIED TEST CASE PRIORITIZATION APPROACH

2.1. Motivation

Our approach is motivated by consideration of the existing additional and total coverage-based test case prioritization strategies. The additional test case prioritization strategy is designed to cover code elements (e.g., statements, branches, or methods) not yet covered by previous test cases, and thus should be well suited for circumstances in which the probability of a test case detecting faults in the program elements it covers is high. The total test case prioritization strategy is designed to cover a maximal number of program elements with each test case, and thus should be well suited for circumstances in which the probability of a test case detecting faults in the program elements it covers is low. Therefore, if we can explicitly consider the probability that a test case will detect faults in the program elements that it covers, we may be able to devise strategies that take advantage of the strengths of both the total and additional strategies.

Our unified approach initially assigns each program element a default probability value that denotes the probability that the element contains a fault. (Henceforth, we use the term “unit” as a generic term to denote the various program elements used in different code-coverage based criteria because our approach is intended to work with any such criteria.) When a test case is executed, it may cover some unit; in doing so it may reveal one or more faults in that unit. The probability that this unit retains undetected faults is thus reduced, and the degree of reduction can be represented by a ratio between 0% and 100%. In essence, the total strategy assumes that this ratio is 0%, whereas the additional strategy assumes that this ratio is 100%. Manipulating this ratio allows us to obtain a spectrum of test case prioritization techniques between the two strategies. That is, our unified approach uses this ratio to incorporate aspects of the additional strategy into the total strategy, and in doing so, we conjecture that it may improve both the total and additional strategies.

2.2. Overview

Our unified test case prioritization approach prioritizes test cases based on the units covered by these test cases and the probabilities that these units contain undetected faults. Consider a test suite $T$ containing $n$ test cases, $\{t_1, t_2, \ldots, t_n\}$, and a program $P$ containing $m$ units $\{u_1, u_2, \ldots, u_m\}$. For any test case $t_i$ and any unit $u_j$, let $Cover[i,j]$ (1 $\leq i \leq n, 1 \leq j \leq m$) denote whether test case $t_i$ covers unit $u_j$, and let $Prob[j]$ (1 $\leq j \leq m$) represent the probability that unit $u_j$ contains undetected faults. Let $Sum[i]$ (1 $\leq i \leq n$) represent the priority weight of test case $t_i$. Our unified test case prioritization approach calculates the priority weight of any test case $t_i$ as follows:

$$Sum[i] = \sum_{j=1}^{m} Prob[j] \times Cover[i,j]$$  \hspace{1cm} (1)

In this equation, $Cover[i,j]$ is 1 if test case $t_i$ covers unit $u_j$, and $Sum[i]$ determines the execution order of test cases in $T$. In particular, the larger the value of $Sum[i]$, the earlier test case $t_i$ should be executed.

When a unit is executed by a test case, faults in the unit may be revealed; execution of a unit thus reduces the probability that the unit contains undetected faults. Following this intuition, we use $f_p$ to represent the probability that test case $t_i$ detects faults in $u_j$ when test case $t_i$ covers unit $u_j$, and $Prob[j]$ to represent the probability that unit $u_j$ contains undetected faults. In particular, as the probability that test case $t_i$ detects faults in $u_j$ ranges from 0 to 1, the value of variable $f_p$ ranges from 0 to 1. During the execution of unit $u_j$, faults in $u_j$ may be detected with probability $f_p$, and
thus the probability $Prob[j]$ associated with unit $u_j$ decreases. That is, our unified test case prioritization approach modifies the value of $Prob[j]$ based on the value of $f_p$. In particular, the larger $f_p$ is, the smaller $Prob[j]$ is after modification.

We define two models by which to modify $Prob[j]$, a basic model and an extended model, respectively. The probability that unit $u_j$ contains undetected faults is dependent, in part, on the extent to which faults in $u_j$ have been covered by previous test cases, which is intuitively related to $f_p$. Therefore, a basic model for $Prob[j]$ modifies $Prob[j]$ based solely on $f_p$. The probability that unit $u_j$ contains undetected faults may also be related, however, to how many times $u_j$ has been covered by a test case. Thus an extended model for $Prob[j]$ modifies $Prob[j]$ based on $f_p$ and the number of times a unit has been covered by a test case. We present details on the two models in Sections 2.3 and 2.4.

Because faults in some units may be easier to detect than faults in other units, the probability that $t_i$ can detect faults in $u_j$ (i.e., $f_p$) should arguably be different for different units. We can extend our approach by assigning differentiated $f_p$ values rather than uniform $f_p$ values. We describe this extension in Section 2.5.

There are two types of coverage information, static coverage information and dynamic coverage information. Dynamic coverage information is more precise than static, and can lead to more effective prioritization results. However, to collect dynamic coverage information, it is necessary to instrument a program under test and run that instrumented program, and this can be costly, or sometimes even impossible. Our unified test case prioritization approach can be applied using either dynamic or static coverage information, and thus allows these cost-benefits tradeoffs to be considered. In the next three sections, for ease of presentation, we present our unified approach relative to dynamic coverage information. Then, in Section 2.6, we describe how the approach can be applied utilizing static coverage information as explored in our previous work [Mei et al. 2012].

2.3. Basic Model

Algorithm 1 depicts our unified test case prioritization approach using the basic model. Initially, we set $Prob[j]$ ($1 \leq j \leq m$) to 1. We use a Boolean array $Selected[i]$ ($1 \leq i \leq n$) to record whether test case $t_i$ has been selected for prioritization. Initially, we set $Selected[i]$ ($1 \leq i \leq n$) to false. We use array $Priority[i]$ ($1 \leq i \leq n$) to store the prioritized test cases. If $Priority[i]$ is equal to $k$ ($1 \leq i, k \leq n$), test case $t_k$ is ordered in the $i$th position.

In Algorithm 1, lines 1-6 perform initialization, and lines 7-39 determine which test case to place in the prioritized test suite. Within the main loop (lines 7-39), lines 8-31 find a test case $t_k$ such that $t_k$ has previously not been chosen and the sum of probabilities that units covered by $t_k$ contain undetected faults is the highest among test cases not yet chosen. In the basic model, $Cover[i,j]$ denotes whether test case $t_i$ covers unit $u_j$. In particular, if test case $t_i$ covers unit $u_j$, $Cover[i,j]$ is 1; otherwise, $Cover[i,j]$ is 0. Because the basic model utilizes a uniform probability $f_p$ for fault detection in covered units, lines 8-31 actually find a test case with the highest probability of detecting previously undetected faults. In particular, lines 8-17 find the first test case $t_k$ not previously chosen for prioritization and calculate the sum of the probabilities that units covered by $t_k$ contain undetected faults, and lines 18-31 determine whether there is another unchosen test case $t_i$ for which the sum of the probabilities that the covered units contain undetected faults is greater than that for $t_k$. Line 32 sets the $i$th position in the prioritized test suite to $t_k$, and line 33 marks $t_k$ as already chosen for prioritization. Lines 34-38 update the probability that units contain undetected faults for each unit covered by $t_k$.
Algorithm 1 Prioritization in the basic model with $f_p$

1: for each $j$ (1 ≤ $j$ ≤ $m$) do
2: $\text{Prob}[j] \leftarrow 1$
3: end for
4: for each $i$ (1 ≤ $i$ ≤ $n$) do
5: $\text{Selected}[i] \leftarrow \text{false}$
6: end for
7: for each $i$ (1 ≤ $i$ ≤ $n$) do
8: $k \leftarrow 1$
9: while $\text{Selected}[k]$ do
10: $k \leftarrow k + 1$
11: end while
12: $\text{sum} \leftarrow 0$
13: for each $j$ (1 ≤ $j$ ≤ $m$) do
14: if $\text{Cover}[k, j]$ then
15: $\text{sum} \leftarrow \text{sum} + \text{Prob}[j]$
16: end if
17: end for
18: for each $l$ (1 ≤ $l$ ≤ $n$) do
19: if not $\text{Selected}[l]$ then
20: $s \leftarrow 0$
21: for each $j$ (1 ≤ $j$ ≤ $m$) do
22: if $\text{Cover}[l, j]$ then
23: $s \leftarrow s + \text{Prob}[j]$
24: end if
25: end for
26: if $s > \text{sum}$ then
27: $\text{sum} \leftarrow s$
28: $k \leftarrow l$
29: end if
30: end if
31: end for
32: $\text{Priority}[i] \leftarrow k$
33: $\text{Selected}[k] \leftarrow \text{true}$
34: for each $j$ (1 ≤ $j$ ≤ $m$) do
35: if $\text{Cover}[k, j]$ then
36: $\text{Prob}[j] \leftarrow \text{Prob}[j] \times (1 - f_p)$
37: end if
38: end for
39: end for

Given Algorithm 1, an optimistic tester who believes that a test case is likely to detect faults in covered units may set the value of $f_p$ to 1. In this case the algorithm encodes a purely additional strategy. The reason for this is that lines 34-38 set the probability for any previously covered unit to contain undetected faults to 0. In contrast, a pessimistic tester who is concerned with the situation in which a test case may not detect faults in covered units may set the value of $f_p$ to 0. In this case this algorithm encodes a purely total strategy. The reason for this is that lines 34-38 do not change the probability that any previously covered unit contains undetected faults. Finally, if a tester sets the value of $f_p$ to a number between 0 and 1, the algorithm encodes a technique between the total and additional strategies. The closer $f_p$ is to 0,
the closer the algorithm is to the total strategy, and the closer \( f_p \) is to 1, the closer the algorithm is to the additional strategy.

Algorithm 1's worst case time cost is the same as that of the additional strategy, namely, \( O(mn^2) \), where \( n \) is the number of test cases and \( m \) is the number of units [Rothermel et al. 1999]. Note that setting \( f_p \) to 0 does not render the algorithm as efficient as the original total strategy, for which the worst case time cost is \( O(mn) \) [Rothermel et al. 1999].

2.4. Extended Model

Intuitively, the more times \( t \) covers \( u \), the more likely it is for \( t \) to detect faults in \( u \). Considering the ability of a given test case to cover a unit multiple times may result in a more effective prioritization approach, so we extend our basic model to consider multiple coverage of units by given test cases.

In our extended model, the main body of the algorithm is the same as the algorithm used in our basic model, but the extended algorithm uses a different method for calculating the probability that a test case detects previously undetected faults. We extend \( \text{Cover}[i, j] \) (\( 1 \leq i \leq n, 1 \leq j \leq m \)) to denote the number of times test case \( t_j \) covers unit \( u_i \). We now describe the main differences between the two algorithms.

First, if we consider each instance of coverage to have an equal probability \( f_p \) of detecting faults, the probability that unit \( u_j \) contains undetected faults changes from \( \text{Prob}[j] \) to \( \text{Prob}[j] \ast (1 - f_p)^{\text{Cover}[k,j]} \) after \( t_k \) is executed. That is to say, for unit \( u_j \) alone, execution of \( t_k \) decreases the probability that \( u_j \) contains undetected faults by \( \text{Prob}[j] \ast (1 - f_p)^{\text{Cover}[k,j]} \). Thus, in our extended algorithm, we change line 15 and 23 of Algorithm 1 to \( \text{sum} \leftarrow \text{sum} + \text{Prob}[j] \ast (1 - f_p)^{\text{Cover}[k,j]} \) and \( s \leftarrow s + \text{Prob}[j] \ast (1 - f_p)^{\text{Cover}[k,j]} \), respectively.

Second, after we select test case \( t_k \) for prioritization at the \( i \)th place, the probability that unit \( u_j \) contains undetected faults following execution of \( t_k \) changes from \( \text{Prob}[j] \) to \( \text{Prob}[j] \ast (1 - f_p)^{\text{Cover}[k,j]} \). Thus, in the extended algorithm, we change line 36 of Algorithm 1 to \( \text{Prob}[j] \leftarrow \text{Prob}[j] \ast (1 - f_p)^{\text{Cover}[k,j]} \).

In the extended algorithm, if we set \( f_p \) to 1, the algorithm is equivalent to a purely additional strategy, because \( (1 - f_p)^{\text{Cover}[k,j]} \) is equal to 0 when \( f_p \) is equal to 1. However, if we set \( f_p \) to 0, the extended algorithm cannot distinguish test cases from each other, because \( 1 - (1 - f_p)^{\text{Cover}[k,j]} \) is always equal to 0 when \( f_p \) is equal to 0. If we set \( f_p \) to a number between 0 and 1, the extended algorithm also represents a strategy between the total and additional strategies, considering multiple coverage for each test case.

The worst case time cost of the extended algorithm is \( O(mn^2) \), the same as that of Algorithm 1.

2.5. Differentiated \( f_p \) Values

In our basic model, whenever a test case \( t \) covers a unit \( u \), we consider the probability that \( t \) will detect faults in \( u \) to be uniformly \( f_p \). In our extended model, we reason that when \( t \) covers \( u \) multiple times, the probability that \( t \) will detect faults in \( u \) may not be uniform, but each instance of coverage also implies a uniform probability of fault detection.

In reality, faults in some units may be easier to detect than faults in other units. Thus, in this section, we further extend our unified approach to account for the situation in which the probability of fault detection varies. To handle this situation, we

\footnote{\text{This limitation is due to the specific algorithm, but our extended model implementation yields the total strategy when } f_p = 0.}
need to assign different probability values that test cases will detect faults in different units. The challenge for performing such an assignment lies in effectively estimating the probability of fault detection. In this work, we estimate differentiated $f_p$ values at the method level based on two widely used static metrics: MLoC, which stands for Method Line of Code, and McCabe, which stands for the well-known McCabe’s Cyclomatic Complexity [McCabe 1976]. Our approach is based on the intuition that methods with larger volume (e.g., higher MLoC values) or greater complexity (e.g., higher McCabe values) need to be covered more times to reveal faults they contain; i.e., they should have lower $f_p$ values. Suppose that the $f_p$ value for each method is in the range $[p_{low}, p_{high}]$, where $0 \leq p_{low} \leq p_{high} \leq 1$, formally, we use both linear normalization (Formula (2)) and log normalization (Formula (3)) to calculate the $f_p$ value for the $j$th method (i.e., $f_p[j]$) as follows:

\[
p_{high} - (p_{high} - p_{low}) \times \frac{\text{Metric}[j] - \text{Metric}_{\text{min}}}{\text{Metric}_{\text{max}} - \text{Metric}_{\text{min}}} \tag{2}
\]

\[
p_{high} - (p_{high} - p_{low}) \times \frac{\log_{10}(\text{Metric}[j] + 1) - \log_{10}(\text{Metric}_{\text{min}} + 1)}{\log_{10}(\text{Metric}_{\text{max}} + 1) - \log_{10}(\text{Metric}_{\text{min}} + 1)} \tag{3}
\]

In these formulas, Metric[j] denotes the MLoC or McCabe metric values for the $j$th method, and Metric_{min}/Metric_{max} denotes the minimum/maximum metric value among all methods of the program under test.\(^2\)

The foregoing metrics and the two formulas for calculating $f_p$ yield four heuristics for generating a differentiated $f_p$ value for each method. In the algorithms for both our basic and extended models, we change all references to the uniform $f_p$ into the differentiated $f_p[j]$ generated for the specific $j$th method. For the basic model, we change line 36 of Algorithm 1 to

\[
\text{Prob}[j] \leftarrow \text{Prob}[j] \times (1 - f_p[j]).
\]

Similarly, for the extended model, we change lines 15, 23, and 36 of Algorithm 1 to

\[
\text{sum} \leftarrow \text{sum} + \text{Prob}[j] \times (1 - (1 - f_p[j])^{\text{Cover}[k,j]}), \quad s \leftarrow s + \text{Prob}[j] \times (1 - (1 - f_p[j])^{\text{Cover}[l,j]}), \quad \text{and} \quad \text{Prob}[j] \leftarrow \text{Prob}[j] \times (1 - f_p[j])^{\text{Cover}[k,j]},
\]

respectively.

The worst case time costs of the basic and extended models with differentiated $f_p$ values remain $O(mn^2)$.

### 2.6. Static Coverage Based Prioritization

Dynamic coverage information is widely used in test case prioritization because it is precise and because test case prioritization based on dynamic coverage information is usually effective. In practice, however, collecting dynamic coverage information can be expensive. To collect dynamic coverage information, it is necessary to instrument the program under test and then run the instrumented program, and this carries overhead. Sometimes it is even infeasible to collect dynamic coverage information. For example, the coverage collection overhead may make real-time systems crash. In such scenarios, it may be necessary to rely on static coverage information in test case prioritization. Therefore, we extend our unified test case prioritization strategy to utilize static coverage information, in a manner similar to that presented in our prior work [Mei et al. 2012].

For Java programs tested in the JUnit framework,\(^3\) test cases are pieces of executable source code containing test content. That is, a JUnit test case is a sequence of method invocations. We can estimate the static coverage (e.g., statement coverage or method coverage) of a JUnit test case based on the methods of source

---

\(^2\)All metric values are increased by 1 in the log normalization to avoid the $\log_{10} 0$ exception.

\(^3\)http://www.junit.org
code under test that are called, directly or indirectly, by the JUnit test case. To do this, we construct a static call graph for a JUnit test case (in this work, by using Jtop\(^4\) [Zhang et al. 2009a]), and we record the methods that may be called directly or indirectly by the JUnit test case by analyzing the call graph. We then estimate static coverage based on these recorded methods. In particular, we use the set of recorded methods to represent the static method coverage of the corresponding test case, and the set of statements that belong to any of these recorded methods to represent the static statement coverage of the corresponding test case. Note that for programs written in other programming/testing paradigm (e.g., C with system tests), we may not be able to extract static coverage for each test since the extracted call graphs for all tests would be the same.

Based on static coverage information, we can conceivably implement our basic model with both uniform \(f_p\) values and differentiated \(f_p\) values. Because static coverage information is collected based on static analysis, however, it can be much less precise than dynamic coverage information, yielding much less effective prioritization techniques than those based on dynamic coverage data. Furthermore, based on the static call graph, it can be difficult to estimate how many times each method is called by a test case, and thus we cannot easily alter the extended model to use static coverage information. We leave the possibility of such an extension to future work.

2.7. Summary
We have presented a unified test case prioritization approach that generates a spectrum of test case prioritization techniques based on model type (basic and extended), \(f_p\) value type (uniform and differentiated), and coverage type (static and dynamic). We treat model type, \(f_p\) value type, and coverage type as three internal factors that may impact the effectiveness of our unified test case prioritization approach. Given the different combinations of the values of these factors, in principal, there are eight different test case prioritization strategies possible, and for each of these eight, there are a range of techniques based on choice of \(f_p\) value.

3. EMPIRICAL STUDY
To evaluate the effectiveness of our unified test case prioritization approach we designed and conducted a controlled experiment, applying the approach to 28 non-trivial Java objects and 40 C objects.

In our experiment, we first compare the proposed unified approach with the total and additional strategies, and then investigate the impact of the three internal factors discussed in Section 2 (i.e., model type, \(f_p\) value, and coverage type) on the effectiveness of our unified approach. We also investigate three potential external factors that may impact the effectiveness of test case prioritization and that have been studied in prior research. These include coverage granularity (the level of code at which coverage is measured), test case granularity (the level of program unit on which test cases are executed), and programming/testing paradigm.

3.1. Research Questions
Our empirical study addresses the following research questions:

— **RQ1**: How do test case prioritization strategies generated by our unified approach compare to the purely total and purely additional strategies?

\(^4\)https://sourceforge.net/projects/pkujtop/
Table I. Statistics on C Objects of Study

<table>
<thead>
<tr>
<th>Object</th>
<th>LoC</th>
<th>MainLoC</th>
<th>#Test</th>
<th>Object</th>
<th>LoC</th>
<th>MainLoC</th>
<th>#Test</th>
</tr>
</thead>
<tbody>
<tr>
<td>base64</td>
<td>3899</td>
<td>105</td>
<td>1915</td>
<td>basename</td>
<td>4026</td>
<td>39</td>
<td>1235</td>
</tr>
<tr>
<td>chcon</td>
<td>4343</td>
<td>195</td>
<td>521</td>
<td>cksum</td>
<td>3983</td>
<td>62</td>
<td>4</td>
</tr>
<tr>
<td>env</td>
<td>3957</td>
<td>45</td>
<td>3074</td>
<td>dircolors</td>
<td>4093</td>
<td>190</td>
<td>778</td>
</tr>
<tr>
<td>expr</td>
<td>9565</td>
<td>338</td>
<td>338</td>
<td>fold</td>
<td>3916</td>
<td>151</td>
<td>780</td>
</tr>
<tr>
<td>groups</td>
<td>4002</td>
<td>37</td>
<td>2177</td>
<td>link</td>
<td>3829</td>
<td>28</td>
<td>628</td>
</tr>
<tr>
<td>logname</td>
<td>3902</td>
<td>25</td>
<td>630</td>
<td>mkdr</td>
<td>4215</td>
<td>66</td>
<td>581</td>
</tr>
<tr>
<td>nice</td>
<td>4010</td>
<td>59</td>
<td>1441</td>
<td>mknode</td>
<td>3840</td>
<td>80</td>
<td>813</td>
</tr>
<tr>
<td>wc</td>
<td>4075</td>
<td>262</td>
<td>17987</td>
<td>unlink</td>
<td>3865</td>
<td>25</td>
<td>608</td>
</tr>
<tr>
<td>unexpand</td>
<td>3903</td>
<td>194</td>
<td>8731</td>
<td>tsort</td>
<td>3856</td>
<td>203</td>
<td>706</td>
</tr>
<tr>
<td>tr</td>
<td>4150</td>
<td>659</td>
<td>713</td>
<td>touch</td>
<td>4744</td>
<td>145</td>
<td>7660</td>
</tr>
<tr>
<td>ttc</td>
<td>3966</td>
<td>69</td>
<td>3051</td>
<td>sync</td>
<td>3919</td>
<td>20</td>
<td>231</td>
</tr>
<tr>
<td>sum</td>
<td>4068</td>
<td>93</td>
<td>3027</td>
<td>split</td>
<td>4428</td>
<td>217</td>
<td>2470</td>
</tr>
<tr>
<td>sleep</td>
<td>4199</td>
<td>46</td>
<td>640</td>
<td>setuid</td>
<td>3875</td>
<td>41</td>
<td>499</td>
</tr>
<tr>
<td>rmdir</td>
<td>3892</td>
<td>72</td>
<td>755</td>
<td>readlink</td>
<td>4154</td>
<td>50</td>
<td>8229</td>
</tr>
<tr>
<td>printf</td>
<td>4251</td>
<td>257</td>
<td>23135</td>
<td>patchchk</td>
<td>3857</td>
<td>132</td>
<td>1140</td>
</tr>
<tr>
<td>paste</td>
<td>3837</td>
<td>187</td>
<td>1075</td>
<td>od</td>
<td>4463</td>
<td>711</td>
<td>582</td>
</tr>
</tbody>
</table>

— **RQ2**: How do the three internal factors (i.e., model type, \( f_p \) value, and coverage type) considered in our unified approach impact the effectiveness of test case prioritization techniques?

— **RQ3**: How do the three external factors (coverage granularity, test case granularity, and programming/testing paradigm) considered impact the effectiveness of test case prioritization techniques?

### 3.2. Objects of Study

To help reduce the threats to external validity that exist when objects of study are homogeneous in nature, we chose to select object programs from two languages, C and Java. For each programming language (i.e., C/Java), we selected object programs that had been provided with test cases of the forms typically used in practice, and made available with those systems. This resulted in the selection of 40 C objects with system test cases, and 28 Java objects with JUnit test cases.

The 40 C objects were selected from version 6.11 of the GNU Core Utilities and implement the basic file, shell, and text manipulation services available on the GNU operating system.\(^5\) We did not use all of the programs from the GNU Core Utilities because some programs do not produce output or their output is related to a specific environmental element (e.g., date), rendering it difficult to determine whether faults in the programs are revealed.

The 28 Java objects are versions of the following programs: Barbecue,\(^6\) a library for creating barcodes; Mime4J,\(^7\) a parser for e-mail message streams in MIME format; Jaxen,\(^8\) which implements the XPath engine; TimeMoney,\(^9\) which manipulates time and money; JDepend,\(^10\) which measures the quality of code design; JodaTime,\(^11\)

\(^{5}\)http://gnu.org/software/coreutils
\(^{6}\)http://barbecue.sourceforge.net/
\(^{7}\)http://james.apache.org/mime4j/
\(^{8}\)http://jaxen.codehaus.org/
\(^{9}\)http://timeandmoney.sourceforge.net/
\(^{10}\)http://clarkware.com/software/JDepend.html
\(^{11}\)http://joda-time.sourceforge.net/
A Unified Test Case Prioritization Approach 39:11

Table II. Statistics on Java Objects of Study

<table>
<thead>
<tr>
<th>Object</th>
<th>Source code under test</th>
<th>JUnit test suite</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>KLoC</td>
<td>#Class</td>
</tr>
<tr>
<td>Barbecue</td>
<td>5.39</td>
<td>59</td>
</tr>
<tr>
<td>Mime4J</td>
<td>6.95</td>
<td>103</td>
</tr>
<tr>
<td>Jaxen</td>
<td>13.9</td>
<td>205</td>
</tr>
<tr>
<td>TimeMoney</td>
<td>2.68</td>
<td>34</td>
</tr>
<tr>
<td>JDepend</td>
<td>2.72</td>
<td>29</td>
</tr>
<tr>
<td>JodaTime</td>
<td>32.9</td>
<td>225</td>
</tr>
<tr>
<td>CommonsLang</td>
<td>23.4</td>
<td>138</td>
</tr>
<tr>
<td>XStream</td>
<td>18.4</td>
<td>347</td>
</tr>
<tr>
<td>Commons-Pool</td>
<td>4.66</td>
<td>50</td>
</tr>
<tr>
<td>Jtopas-v1</td>
<td>1.89</td>
<td>19</td>
</tr>
<tr>
<td>Jtopas-v2</td>
<td>2.03</td>
<td>21</td>
</tr>
<tr>
<td>Jtopas-v3</td>
<td>5.36</td>
<td>50</td>
</tr>
<tr>
<td>Xml-security-v1</td>
<td>18.3</td>
<td>179</td>
</tr>
<tr>
<td>Xml-security-v2</td>
<td>19.0</td>
<td>180</td>
</tr>
<tr>
<td>Xml-security-v3</td>
<td>16.9</td>
<td>145</td>
</tr>
<tr>
<td>JMeter-v1</td>
<td>33.7</td>
<td>334</td>
</tr>
<tr>
<td>JMeter-v2</td>
<td>33.1</td>
<td>319</td>
</tr>
<tr>
<td>JMeter-v3</td>
<td>37.3</td>
<td>373</td>
</tr>
<tr>
<td>JMeter-v4</td>
<td>38.4</td>
<td>380</td>
</tr>
<tr>
<td>JMeter-v5</td>
<td>41.1</td>
<td>389</td>
</tr>
<tr>
<td>Ant-v1</td>
<td>25.8</td>
<td>228</td>
</tr>
<tr>
<td>Ant-v2</td>
<td>39.7</td>
<td>342</td>
</tr>
<tr>
<td>Ant-v3</td>
<td>39.8</td>
<td>342</td>
</tr>
<tr>
<td>Ant-v4</td>
<td>61.9</td>
<td>532</td>
</tr>
<tr>
<td>Ant-v5</td>
<td>63.5</td>
<td>536</td>
</tr>
<tr>
<td>Ant-v6</td>
<td>63.6</td>
<td>536</td>
</tr>
<tr>
<td>Ant-v7</td>
<td>80.4</td>
<td>649</td>
</tr>
<tr>
<td>Ant-v8</td>
<td>80.4</td>
<td>650</td>
</tr>
</tbody>
</table>

which implements functions similar to standard Java date and time classes; CommonsLang,\(^{12}\) which extends the standard Java library; XStream,\(^{13}\) a library for fast serialization/deserialization to/from XML; Commons-Pool,\(^{14}\) which provides a concurrent object-pooling API; three versions of Jtopas,\(^{15}\) a Java library for parsing arbitrary text data; three versions of Xml-security,\(^{16}\) a component library implementing XML signature and encryption standards; five versions of JMeter,\(^{17}\) a Java desktop application; and eight versions of Ant,\(^{18}\) a Java-based build tool. Note that we obtained versions of Jtopas, Xml-security, JMeter, and Ant from Software-artifact Infrastructure Repository (SIR\(^{19}\)) that has been widely used in software testing research [Do et al. 2004; Hsu and Orso 2009; Zhang et al. 2011; Gligoric et al. 2013; Zhang et al. 2013; Zhang et al. 2014], while all the other objects are from their own repository.

Tables I and II provide basic statistics on these objects. In Table I, “LoC” denotes the total number of lines of code for each object including header files and any library files it uses (excluding white spaces and comments); “MainLoC” denotes the number of lines of code for each object’s main C files (further excluding headers and library

\(^{12}\)http://commons.apache.org/proper/commons-lang/
\(^{13}\)http://xstream.codehaus.org/
\(^{14}\)http://commons.apache.org/pool/
\(^{15}\)http://jtopas.sourceforge.net/jtopas
\(^{16}\)http://xml.apache.org/security
\(^{17}\)http://jakarta.apache.org/jmeter
\(^{18}\)http://ant.apache.org
\(^{19}\)http://sir.unl.edu/
files), and “#Test” denotes the total number of test cases used in system testing for each object.

Each of the objects of study is provided with a test suite that was either constructed during the object’s evolution (e.g., for all the Java objects) or automated generated using state-of-the-art tool (e.g., the tests for C objects are generated using KLEE [Cadar et al. 2008]). Because previous research [Andrews et al. 2005; Andrews et al. 2006; Do and Rothermel 2006b; Just et al. 2014] has confirmed that it is suitable to use faults produced via mutation for experimentation in test case prioritization, we followed a similar process to produce faulty versions for each program. In particular, we used MutGen [Andrews et al. 2005] to generate mutation faults for C object programs. We used Javalanche [Schuler and Zeller 2009] to generate mutation faults for the first 9 Java object programs. Similar to previous work, we used MuJava21 [Ma et al. 2005] to generate mutation faults for the remaining Java object programs. We tried to use one mutation tool for all subjects. However, Javalanche tends to generate infinitely large temporary files for Ant and not terminate, while MuJava throws various exceptions for some objects from the first set. Given that the first set of 9 objects come from open-source repositories, while the other 19 come from the SIR repository, we finally chose to use the same tool across each set of objects (i.e., Javalanche for the first set and MuJava for the second set).

3.3. Independent Variables

Our study considers the following four independent variables.

Prioritization Strategy. As described in Section 2.7, our unified test case prioritization approach generates a spectrum of prioritization techniques based on various values of three internal factors: model type, \( f_p \) values, and coverage type. Our study varied and combined the values of these factors to define several techniques. Where model type is concerned, we used the basic and extended models. For \( f_p \) values, we used two types of \( f_p \) values: uniform and differentiated. We implemented the unified approach using the basic and extended models with uniform \( f_p \) values ranging from 0.05 to 0.95 in increments of 0.05, and differentiated \( f_p \) values calculated by one of four heuristics (the MLoC metric and linear normalization, the MLoc metric and log normalization, the McCabe metric and linear normalization, and the McCabe metric and log normalization). When implementing the unified approach with differentiated \( f_p \) values, we restricted \( f_p \) values to be between 0.5 and 1.0 (i.e., \( p_{low} \) is set to 0.5 and \( p_{high} \) is set to 1.0 in Formulas 2 and 3).22 For coverage type, we implemented the unified approach for both static and dynamic coverage.

In our empirical study, we focus on the five test case prioritization strategies listed in Table III. As noted earlier, it is difficult to estimate how many times each unit can be covered by a test case statically, we do not consider strategies that use the extended model and static coverage in this study. The study also omits the strategy that utilizes the basic model and static coverage with differentiated \( f_p \) values for each covered unit. We omit this latter strategy because when prioritizing test cases using static coverage, a unit determined as “covered” by a test case may actually not be

---

20 Andrews et al. [Andrews et al. 2005; Andrews et al. 2006] also reported empirical evidence for the suitability of using mutation faults in general testing experiments.

21 http://cs.gmu.edu/~offutt/mujava

22 We do this for two reasons. First, according to our previous work and this experimental study on both real-world C and Java programs, the additional strategy (i.e., \( f_p = 1.0 \) in our models) is usually much more effective than the total strategy (i.e., \( f_p = 0.0 \) in our models), indicating that test cases are usually relatively good at revealing faults. Second, according to our study on uniform \( f_p \) values from 0.0 to 1.0 for both our basic and extended models, \( f_p \) values close to 1.0 are usually more effective than \( f_p \) values close to 0.0, indicating that higher \( f_p \) values are usually better for test case prioritization.
covered, rendering a precise probabilistic calculation for each unit unnecessary for such an imprecise baseline technique.

As controlled techniques for use in comparisons, we use techniques based on the total and additional strategies.

In our study, we apply our unified test case prioritization approaches utilizing static coverage only to Java programs, not C programs. The static call graphs for test cases of C programs include all methods in the program under test, and thus it is less effective to prioritize test cases for such programs based on their static call graphs. We will investigate how to effectively prioritize test cases for C programs utilizing static coverage in the future. Further, we apply our unified approach utilizing differentiated \( f_p \) values only to the Java programs.

**Coverage Granularity.** In prior research on test case prioritization [Mei et al. 2012], researchers treated coverage granularity as a constituent part of prioritization techniques. Because our goal is to investigate various unified prioritization strategies, we treat coverage granularity separately. We consider structural coverage criteria at both the method level and the statement level. Note that we consider differentiated \( f_p \) values only at the method level, because our current metrics focus on method complexity, and we have not been able to find or define reasonable or well-established metrics that can calculate complexity at the statement level.

**Test Case Granularity.** The Java objects we use in our study have two levels (granularities) of test cases: test-class and test-method. To investigate the impact of test case granularity on our prioritization approach, we consider both of these test case granularity levels. For the test-class level each JUnit TestCase class is treated as a test case, whereas for the test-method level each test method in a JUnit TestCase class is treated as a test case.

**Programming and Testing Paradigm.** To investigate the impact of programming and testing paradigm on our prioritization approach, we consider the two programming languages that our object programs utilized, Java and C. Furthermore, we utilize testing paradigms that are most widely used for these two programming languages; that is, for Java programs we utilize unit testing in the JUnit testing framework and for C programs we utilize system testing.

### 3.4. Dependent Variable

Our dependent variable tracks technique effectiveness. We adopt the well-known APFD (Average Percentage Faults Detected) metric [Rothermel et al. 1999]. The APFD metric is widely used in studies of test case prioritization, such as those presented in Elbaum et al. [Elbaum et al. 2000; Elbaum et al. 2002], Do and Rothermel [Do et al. 2004; Do and Rothermel 2006b], and Jiang et al. [Jiang et al. 2009]. It tracks the rate of fault detection of test suites.
Let $T$ be a test suite and $T'$ be a permutation of $T$, the APFD for $T'$ is defined as follows:

$$APFD = \left( \frac{\sum_{i=1}^{n-1} F_i}{n \cdot l} + \frac{1}{2n} \right) * 100\%$$ (4)

Here, $n$ is the number of test cases in $T$, $l$ is the number of faults, and $F_i$ is the number of faults detected by at least one test case among the first $i$ test cases in $T'$.

3.5. Experiment Process

For each object program, we employed the following experiment process.

We followed the process described by Do et al. [Do and Rothermel 2006b] to select specific mutants and mutant groups to use because in actual testing scenarios the number of faults contained in a specific program version is bounded. Thus, we constructed multiple-fault programs by grouping mutants generated by MutGen, MuJava, or Javalanche. Specifically, to create one mutant group, we randomly selected five mutants that could be killed by at least one test case in the test suite for the program. Then we created up to 20 mutant groups following the same process, while ensuring that no mutant was used in more than one group.

Second, we collected static coverage information for each Java object program and applied the static coverage based unified test case prioritization approach to those programs. In particular, we used Jtop [Mei et al. 2012] to construct a static call graph for each test case (either at the test-class level or the test-method level) so as to determine its static method coverage. Next, we used each mutant group to create a potential subsequent version of the program containing those mutations, and applied all static coverage based test case prioritization techniques to each test suite for those subsequent versions, recording which faults were detected by each test case.

Third, we collected dynamic coverage information for each object program and applied each dynamic coverage based test case prioritization technique to each program. To collect dynamic coverage information for Java objects, we used on-the-fly byte-code instrumentation, which dynamically instruments classes loaded into the JVM through a Java agent without any modification of the target program. We implemented code instrumentation based on the ASM byte-code manipulation and analysis framework. In particular, we inherited the visitor classes defined in the ASM framework, and added our code for recording coverage information. To collect dynamic coverage information for C objects, we used the “gcov” command of GCC. To simplify the implementation of our approach, we extended the Eclipse AST analysis and parsing tool to calculate MLoC and McCabe values while implementing our unified approach with differentiated $f_p$ values.

Finally, given a program version $V$ and a prioritization technique $S$ with a model type $M_u$, a coverage granularity level $C_m$, a coverage type $Y_n$, a test case granularity level $T_k$ and a method for calculating $f_p$ values denoted by $p_i$, we obtained the effectiveness of technique $S$ on $V$ for $M_u$, $C_m$, $Y_n$, and $T_k$ as follows.

23Because Jmeter-v1 has only 35 mutants that can be killed by at least one test case, we produced only seven mutant groups for this object.

24http://asm.ow2.org
25http://www.eclipse.org/jdt/
26Our JavaSourceMetric tool is available at http://sourceforge.net/projects/jsourcemetric/.
We calculated the APFD values of the prioritized sequence of test cases for each mutant group of \( V \) at \( M_u, C_m, Y_n, T_k \), and \( p_i \). These values serve as our data sets for analysis.

We followed the same process for the baseline techniques, additional and total, considering relevant combinations of coverage granularity level, coverage type, and test granularity level. (Note that the Java object programs we utilized have two test case granularity levels, whereas the C programs have only one test case granularity level.)

### 3.6. Threats to Validity

Our object programs, test cases, and seeded faults may all pose threats to external validity. The C and Java programs we use are of various sizes and complexity, but we cannot say that they are representative of any particular population of C and Java programs. Furthermore, our results may not generalize to programs written in languages other than Java and C. Second, although we used a procedure that previous research [Andrews et al. 2005; Andrews et al. 2006; Do and Rothermel 2006b; Hao et al. 2013; Hao et al. 2012] has demonstrated to be suitable to produce faulty versions for each program, our results based on programs with seeded (mutation) faults may not be generalizable to programs with real faults or faults generated from other mutation tools (e.g., MAJOR [Just et al. 2011; Just 2014]). Third, although we used test cases distributed with the object programs, our results may not be generalizable to other forms of test cases. To address these threats, further studies involving additional object programs, test suites, and faults are needed.

The main threat to internal validity for our study is the possibility of faults in our implementation of techniques and the calculation of APFD values. To reduce this threat, we carefully reviewed and tested all of the code that we produced before conducting the experiments. Furthermore, we implemented our approach by restricting some parameters’ values (e.g., \( f_p \), \( p_{low} \), and \( p_{high} \)), and we may reduce this threat by further evaluation on our approach with various values of these parameters.

Where construct validity is concerned, to assess technique effectiveness our study used the APFD metric [Rothermel et al. 1999] that is widely used for test case prioritization. However, the APFD metric does have limitations [Do and Rothermel 2006b; Rothermel et al. 1999], and our study did not consider efficiency or other cost and savings factors. Reducing this threat also requires additional studies, using more sophisticated cost-benefit models.

### 4. RESULTS AND ANALYSIS

The following sections present the results of our empirical study relative to our research questions. The data set gathered in this study is available online\(^\text{27}\).

Due to the large number of test case prioritization techniques, test case granularities, coverage granularities, objects, and mutant groups considered in our study, boxplots across all objects provide a suitable way to present descriptive statistics about our results. Figures 1 through 8 use boxplots to summarize the results obtained by applying our test case prioritization techniques to Java object programs using the basic and extended models with uniform \( f_p \) values, along with the results of the total and additional strategies. All of the techniques whose results are depicted prioritize test suites at the test-method or test-class level using static or dynamic coverage of methods or statements.

Similarly, Figures 9 and 10 summarize the results obtained by applying our test case prioritization strategies to C object programs, using the basic and extended mod-

\(^{27}\)https://sites.google.com/site/unifiedtestprioritization
results of these tests for Java programs, and Tables 4.1. RQ1: Comparison with the Total and Additional Strategies

Considering the data presented in the boxplots, prioritization techniques derived using our basic and extended models achieved higher median APFD values than the additional strategy for many \( f_p \) values. This occurred when using static or dynamic coverage information. In particular, for Java programs (Figures 1 through 8), techniques with \( f_p \) values between 0.60 and 0.95 usually achieved mean APFD values greater than their corresponding additional techniques, and for C programs (Figures 9 and 10), techniques with \( f_p \) values between 0.80 and 0.95 achieved greater mean APFD values than their corresponding additional techniques.

To determine whether the results observed in the boxplots are statistically significant, we used SPSS\(^{28}\) to perform Fisher’s Least Significant Difference (LSD) tests\(^{29}\) between pairs of techniques at the 0.05 significance level. Tables IV and V list the results of these tests for Java programs, and Tables VI and VII list the results for C programs. In the tables, the first column (CT) denotes the type of coverage information used, the second column (TCG) denotes test case granularity, the third column (CG) denotes coverage granularity, and subsequent columns correspond to techniques derived using our approach. Entries of “1” indicate cases in which our techniques are statistically significantly better than the corresponding additional technique, entries of “0” indicate cases in which there is no significant difference, and entries of “-1” indicate cases in which our techniques are statistically significantly worse than the additional technique, based on APFD values achieved on individual mutant groups.

28http://www-01.ibm.com/software/analytics/spss/
29LSD is a pairwise comparison technique [Hayter 1986] that is effective for computing the significant differences between two means based on a two-step testing procedure; we chose this test for its power.
As the results show, additional techniques usually did not significantly outperform techniques generated by our unified approach using various model types, coverage
types, and coverage granularities, except when \( f_P \) values were relatively small (e.g., 0.05 – 0.45). Techniques with small \( f_P \) values were closer in terms of performance to \textit{total} techniques. In other cases, techniques generated based on the unified approach were usually not significantly different from \textit{additional} techniques. The primary exception to this involves techniques derived from the unified approach based on dynamic method coverage information, which sometimes (e.g., for the basic model with \( f_P \) values ranging from 0.50 to 0.95 and for the extended model with \( f_P \) values ranging from 0.25 to 0.95) significantly outperformed the \textit{additional} technique when prioritizing JUnit test cases at the test-method level.

Thus, overall, the average increases in APFD for our techniques over techniques following the \textit{additional} strategy were typically relatively small and typically not statistically significant. However, given that our techniques are no more expensive than the \textit{additional} strategy, and that the techniques do at times outperform it, their application can still be worthwhile.

Compared to the \textit{total} technique, prioritization techniques derived using our unified approach outperformed \textit{total} techniques across most \( f_P \) values. This occurred with both the basic and extended models, using both static and dynamic coverage information. Even when the \( f_P \) values were 0.05 (which results in prioritization orders relatively similar to those produced by \textit{total} techniques), techniques derived using our approach were usually more effective. Furthermore, we performed LSD tests between pairs of techniques at the 0.05 significance level, with results shown in Tables VIII to XI.
From these tables, it is clear that techniques derived from the unified approach based on dynamic method coverage information or dynamic statement coverage information usually significantly outperformed the total technique, and techniques derived from the unified approach based on static method coverage information or static statement coverage usually significantly outperformed the total technique when prioritizing JUnit test cases at the test-method level.

In summary, our unified test case prioritization approach generated a spectrum of techniques that were more effective than the total strategy and competitive with the additional strategy. In particular, the unified test case prioritization approach using
Dynamic CT Results of LSD Tests of C Programs Comparing our Strategies Using the Basic Model to the Total Strategy

<table>
<thead>
<tr>
<th>CT</th>
<th>CG</th>
<th>B95</th>
<th>B90</th>
<th>B85</th>
<th>B80</th>
<th>B75</th>
<th>B70</th>
<th>B65</th>
<th>B60</th>
<th>B55</th>
<th>B50</th>
<th>B45</th>
<th>B40</th>
<th>B35</th>
<th>B30</th>
<th>B25</th>
<th>B20</th>
<th>B15</th>
<th>B10</th>
<th>B05</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dynamic</td>
<td>Method</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>Statement</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

Table XI. Results of LSD Tests of C Programs Comparing our Strategies Using the Extended Model to the Total Strategy

<table>
<thead>
<tr>
<th>CT</th>
<th>CG</th>
<th>E95</th>
<th>E90</th>
<th>E85</th>
<th>E80</th>
<th>E75</th>
<th>E70</th>
<th>E65</th>
<th>E60</th>
<th>E55</th>
<th>E50</th>
<th>E45</th>
<th>E40</th>
<th>E35</th>
<th>E30</th>
<th>E25</th>
<th>E20</th>
<th>E15</th>
<th>E10</th>
<th>E05</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dynamic</td>
<td>Method</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>Statement</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

either the basic model or the extended model when \( f_p \) values range from 0.50 to 0.90 was significantly better than the total and additional techniques when prioritizing JUnit test cases at the test-method level using method coverage.

4.2. RQ2: Internal Factor Impact

Our second research question concerns the impact of model type, \( f_p \) value, and coverage type, which are internal factors within our unified approach that may affect the effectiveness of unified test case prioritization.

4.2.1. Impact of Model Type. Comparing techniques derived from our basic model to those derived from our extended model, we find that techniques performed similarly when \( f_p \) values were close to 1 and differently when \( f_p \) values were close to 0. When \( f_p \) values were close to 1, techniques derived from both models achieved APFD values comparable to and even higher than those achieved by corresponding additional techniques. When \( f_p \) values were close to 0, however, techniques derived from the extended model remained competitive but techniques derived from the basic model became less competitive. In other words, techniques using small \( f_p \) values derived from the basic model performed more like total techniques, but techniques derived from the extended model performed like additional techniques with any \( f_p \) values.

To confirm these observations, we performed LSD tests (at the 0.05 level) comparing techniques derived from our basic model to those derived from our extended model. The results are shown in Tables XII and XIII. Entries in the table indicate whether a technique with a particular \( f_p \) value derived from the basic model is significantly better than the corresponding technique derived from the extended model, using the same notation conventions utilized in prior tables.

The data in the tables shows that the differences between techniques derived from the basic and extended models, at given \( f_p \) values, were usually not significant. Exceptions exist primarily in the results for Java programs when \( f_p \) values were close to 0. In these cases, techniques derived from the basic model were more like the total strategy and thus more likely to be less effective than other strategies. The only other exception occurs on the C programs, where the extended model outperforms the basic model when \( f_p = 0.45 \) and \( f_p = 0.50 \) but does not outperform the basic model on other values. We found that both the basic and extended models revealed the faults within a small ratio of prioritized tests, i.e., both models performed quite well for the C programs. The reason for this is that both models were able to cover the mutation faults (which are easy to reveal when covered) within a small ratio of prioritized tests given the additional flavor (i.e., selecting tests covering as-yet uncovered elements) of the models as well as the large number of generated tests for C programs. Therefore, the fact that the extended model outperformed the basic model only when \( f_p = 0.45 \) or \( f_p = 0.50 \) is coincidental because both models performed well for the studied C programs.

In summary, techniques derived using our basic model were similar in performance to techniques derived using our extended model when \( f_p \) values were close to 1, but the latter were more effective when \( f_p \) values were close to 0.
Table XII. Results of LSD Tests on Java Programs Comparing Strategies Using the Basic Model to Strategies Using the Extended Model

<table>
<thead>
<tr>
<th>Method</th>
<th>Statement</th>
<th>B95</th>
<th>B90</th>
<th>B85</th>
<th>B80</th>
<th>B75</th>
<th>B70</th>
<th>B65</th>
<th>B60</th>
<th>B55</th>
<th>B50</th>
<th>B45</th>
<th>B40</th>
<th>B35</th>
<th>B30</th>
<th>B25</th>
<th>B20</th>
<th>B15</th>
<th>B10</th>
<th>B05</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dynamic</td>
<td>Method</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>Statement</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

Table XIII. Results of LSD Tests on C Programs Comparing Strategies Using the Basic Model to Strategies Using the Extended Model

<table>
<thead>
<tr>
<th>Method</th>
<th>Statement</th>
<th>B95</th>
<th>B90</th>
<th>B85</th>
<th>B80</th>
<th>B75</th>
<th>B70</th>
<th>B65</th>
<th>B60</th>
<th>B55</th>
<th>B50</th>
<th>B45</th>
<th>B40</th>
<th>B35</th>
<th>B30</th>
<th>B25</th>
<th>B20</th>
<th>B15</th>
<th>B10</th>
<th>B05</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dynamic</td>
<td>Method</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>Statement</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

4.2.2. Impact of $f_p$ Values. Based on Figures 1 through 10, we make the following observations about the performance of our unified approach using uniform $f_p$ values. As we increased the values of $f_p$ in either the basic or extended models, for Java programs, the generated techniques often achieved higher mean APFD values for Java programs. For example, when using our basic model to prioritize test cases at the test-method level for Java programs using dynamic method coverage, as $f_p$ values increased from 0.05 to 0.50, the average APFD results increased from 79.33 to 89.14. In contrast, when the $f_p$ values increased from 0.50 to 0.95, the average APFD results remained around 89.20 with only slight changes. We derive similar observations from several of the other cases in which Java programs were involved. That is, when $f_p$ values increased from 0 to a medium value, the average APFD results increased, whereas when $f_p$ values increased from a medium value to 0.95, the average APFD results tended to change little.

A similar trend occurred for C programs, when using the basic model to prioritize test cases for C programs using dynamic method coverage (Figure 9). Results of the unified approach using dynamic statement coverage for C programs (Figure 10), in contrast, reveal small increases in APFD values as the $f_p$ values increase. For C programs in general, however, the APFD results of all techniques except for total techniques using dynamic statement coverage were all close to 1. This observation is as expected because the test suites for the C programs contained some high-quality test cases that could reveal relatively large numbers of faults. Furthermore, techniques derived from the unified approach with large $f_p$ values (0.85 to 0.95) were usually effective.

Figure 11 depicts results obtained by comparing techniques using differentiated $f_p$ values with the corresponding additional techniques. We use $BP$ to denote the four techniques derived from the basic model and $EP$ to denote the four techniques derived from the extended model. For the basic model, $BP1$ denotes the use of the MLoC metric and linear normalization, $BP2$ denotes the use of the MLoC metric and log normalization, $BP3$ denotes the use of the McCabe metric and linear normalization, and $BP4$ denotes the use of the McCabe metric and log normalization. Techniques derived from the extended model are labeled following a similar convention.

All techniques with differentiated $f_p$ values outperformed the corresponding additional techniques based on method coverage. For example, when prioritizing test-class level test cases using method coverage, the additional strategy achieved an APFD value of 77.27 on average, while the four techniques derived from the extended model achieved APFD values from 80.19 to 80.78. Furthermore, considering Figures 1 through 4, techniques with uniform $f_p$ values were less effective than techniques with differentiated $f_p$ values. Additional analysis of the comparison between techniques with uniform and differentiated $f_p$ values and techniques with differentiated $f_p$ values can be found in our prior work [Zhang et al. 2013].

As noted earlier, in this empirical study, we consider differentiated $f_p$ values only at the method level.
Fig. 11. Prioritization results for models utilizing differentiated $f_p$ values for each method, at the test-class level (left) and the test-method level (right)

Table XIV. Results of LSD Tests on Java Programs Comparing Strategies Using Static Coverage to Strategies Using Dynamic Coverage

| TCG | CG | B95 | B90 | B85 | B80 | B75 | B70 | B60 | B65 | B50 | B55 | B50 | B45 | B40 | B35 | B30 | B25 | B20 | B15 | B10 | B05 |
|-----|----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|
| Test-Method | Method | -1 | -1 | -1 | -1 | -1 | -1 | -1 | -1 | -1 | -1 | -1 | -1 | -1 | -1 | -1 | -1 | -1 | -1 | -1 | 0 |
| Test-Class | Statement | -1 | -1 | -1 | -1 | -1 | -1 | -1 | -1 | -1 | -1 | -1 | -1 | -1 | -1 | -1 | -1 | -1 | -1 | -1 | 0 |

4.2.3. Impact of Coverage Type. Figures 1 through 10 suggest that, when prioritizing test cases using any model with any $f_p$ value, our unified approach based on dynamic coverage usually outperformed the corresponding unified approach based on static coverage. Furthermore, the unified approach based on static coverage using any model with large $f_p$ values tended to be as effective as the unified approach based on dynamic coverage using the corresponding model with small $f_p$ values, but the former was usually less effective than the unified approach based on dynamic coverage using the corresponding model with the same $f_p$ value. This observation is as expected, because static coverage is less precise than dynamic coverage and the unified test case prioritization approach based on static coverage is thus less effective than the unified approach based on dynamic coverage.

We performed LSD tests (at the 0.05 level) to compare techniques based on static and dynamic coverage. The results are shown in Table XIV.

As the table shows, our unified approach using the basic model and dynamic coverage usually significantly outperformed the unified approach using the basic model and static coverage. However, when prioritizing test cases at the test-class level, the unified approach based on the basic model and dynamic coverage did not significantly outperform the unified approach based on the basic model and static coverage as $f_p$ values ranged from 0.05 to 0.15. Furthermore, when prioritizing test cases at the test-class level, the unified approach based on the basic model and static coverage even significantly outperformed the unified approach based on the basic model and dynamic coverage when $f_p$ values were 0.05. The reason for this observation may be that test cases at the test-class level tend to cover more program units than those at the test-method level, and the effects of the approximations inherent in calculating static coverage are muted in the presence of test cases that cover more program units.

4.3. RQ3: External Factor Impact

Our third research question concerns the impact of coverage granularity, test case granularity, and programming language on unified test case prioritization.

4.3.1. Impact of Coverage Granularity. Our unified test case prioritization approach appears to have been more effective when using coverage information at the method level than at the statement level. In Figures 1 through 10, for both our basic and extended
Table XV. Results of LSD Tests on Java Programs Comparing our Strategies Using the Basic Model and Method Coverage to the Additional Strategy Using Statement Coverage

| TCG  | CT       | E05  | E10  | E15  | E20  | E25  | E30  | E35  | E40  | E45  | E50  | E55  | E60  | E65  | E70  | E75  | E80  | E85  | E90  | E95  |
|------|----------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|
| Test-Class | Static | 0    | 0    | 0    | 0    | 0    | 0    | 0    | 0    | 0    | 0    | 0    | 0    | 0    | -1   | -1   | -1   | -1   | -1   | -1   | -1   |
| Test-Class | Dynamic | 0    | 0    | 0    | 0    | 0    | 0    | 0    | 0    | 0    | 0    | 0    | 0    | 0    | 0    | 0    | 0    | 0    | 0    | 0    | 0    |

Table XVI. Results of LSD Tests on Java Programs Comparing our Strategies Using the Extended Model and Dynamic Method Coverage to the Additional Strategy Using Dynamic Statement Coverage

| TCG  | E05  | E10  | E15  | E20  | E25  | E30  | E35  | E40  | E45  | E50  | E55  | E60  | E65  | E70  | E75  | E80  | E85  | E90  | E95  |
|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|
| Test-Class | Static | -1   | -1   | -1   | -1   | -1   | -1   | -1   | -1   | -1   | -1   | -1   | -1   | -1   | -1   | -1   | -1   | -1   | -1   | -1   | -1   |
| Test-Class | Dynamic | 0    | 0    | 0    | 0    | 0    | 0    | 0    | 0    | 0    | 0    | 0    | 0    | 0    | 0    | 0    | 0    | 0    | 0    | 0    | 0    |

In summary, because method coverage is usually much less expensive to achieve than statement coverage, and because our unified approach based on method coverage produced a large number of test case prioritization techniques that outperformed the corresponding additional techniques, the approach based on method coverage can serve as a less expensive alternative for the additional strategy using statement coverage.

4.3.2. Impact of Test Case Granularity. All the techniques that we considered achieved significantly greater average APFD values when prioritizing test cases at the test-method level than when prioritizing test cases at the test-class level. In fact, for any object and strategy, using either statement coverage or method coverage, the average APFD value when prioritizing test cases at the test-method level was uniformly higher than that for prioritizing test cases at the test-class level. We suspect the reason for this to be that, because a test case at the test-class level consists of a number of test
cases at the test-method level, it is more flexible to prioritize test cases at the test-method level.

Furthermore, our extended model appears to have been more beneficial than our basic model when prioritizing test cases at the test-class level. When using the extended model instead of the basic model, the number of strategies that outperformed the additional strategies increased more dramatically at the test-class level than at the test-method level (as shown in Figures 1 through 8). We suspect the reason for this to be that it is more common for a test case at the test-class level to cover a method or a statement more than once than it is for a test case at the test-method level to do so. In such a circumstance, it is more beneficial to consider information tracking multiple instances of coverage.

4.3.3. Impact of Programming/Testing Paradigm. Figures 1 through 8 show that the APFD results of our unified test case prioritization approach ranged from 50% to 100%, whereas Figures 9 and 10 show that the APFD results of the unified test case prioritization approach ranged from 45% to 100%. However, when prioritizing test cases using dynamic method coverage, most APFD values produced by the approach on Java programs with their unit tests ranged from 70% to 90%, whereas most APFD values produced by the approach on C programs with their system tests ranged from 80% to 100%. When prioritizing test cases using dynamic statement coverage, most results of the approach on Java programs with unit tests ranged from 70% to 95%, whereas most results of the approach on C programs with system tests ranged from 98% to 100%. Therefore, the unified approach was more effective when prioritizing test cases for C programs with system tests than for Java programs with unit tests. Furthermore, Tables IV through XI suggest that the unified approach was less effective on C programs with system tests than on Java programs with unit tests when compared to the total and additional strategies. That is, for all test case prioritization approaches (including the unified approach, the total and additional strategies) in this empirical study produced better results for C programs with system tests than for Java programs with unit tests, but the benefit of applying the approach to C programs with system tests was not as large as that for Java programs with unit tests.

This observation is as expected for the following reasons. System testing aims to test the whole program, and its test cases may cover similar portions of the systems' code. Because test cases covering similar code cannot be distinguished based on the probability calculation defined in our unified approach, the approach may achieve little benefit from incorporating probabilities. Moreover, the C programs are relatively small in size and have a large number of generated tests\(^{31}\) which cover most elements in the programs an extremely large number of times. Therefore, the disadvantage of the additional strategy (i.e., it may not consider some faulty statement after covering it once) does not matter as much here, because after a statement is covered, the statement may be quickly covered again when all the statements have been covered.

4.4. Summary and Implications

We summarize the main findings of our empirical study:

— Our unified test case prioritization approach generated a spectrum of techniques that were more effective than techniques associated with the total strategy and competitive with techniques associated with the additional strategy. Some techniques generated by our unified approach even outperformed the additional strategy.

\(^{31}\)The tests for these C programs were generated automatically using KLEE [Cadar et al. 2008].
Techniques derived from our extended model were similar in performance to techniques derived from our basic model when $f_p$ values were close to 1, but the former were more effective when $f_p$ values were close to 0.

Techniques with differentiated $f_p$ values outperformed techniques with uniform $f_p$ values. Moreover, techniques with large uniform $f_p$ values tended to be more effective than techniques with small uniform $f_p$ values.

Our unified approach with the basic model was more effective when using dynamic coverage information than when using static coverage information. However, these two approaches sometimes exhibited no significant differences when using test cases at the test-class level, in particular when using method coverage.

Our unified approach was more beneficial when using method coverage than when using statement coverage, considering the cost and effectiveness of the unified approach.

Our unified approach was more effective when applied to test cases at the test-method level than at the test-class level.

Our unified approach was more effective for Java programs than C programs in incorporating the benefits of the additional strategy.

The experimental findings provide implications for practitioners. The need for more and better blended approaches provides implications for researchers.

5. DISCUSSION

We now discuss some additional issues related to our unified approach.

First, variable $f_p$ in our unified approach is defined as the probability that test case $t_i$ can detect faults in unit $u_j$. Note that the probability that a test case can detect faults in a unit is different than the probability that a unit can contain faults, although these two probabilities are likely to be related to each other (e.g., a method including more complex structures may be more prone to contain faults and those faults may also be relatively difficult to detect). Various researchers (e.g., Ostrand et al. [Ostrand et al. 2005] and Nagappan et al. [Nagappan et al. 2006]) have proposed techniques with which to predict the probability that a unit contains faults, and cost-based test case prioritization techniques [Elbaum et al. 2001; Malishevsky et al. 2006] can account for these probabilities by differentially weighting fault-prone units.

It is difficult, however, to estimate the probability of detecting faults because that probability depends both on whether the faulty unit has been executed by a test case and on the probability that the fault will cause an observable failure. In fact, the only work that we are aware of that has attempted to use such predictions in the context of prioritization is presented by Elbaum et al. [Elbaum et al. 2002], in which estimates of the fault-exposing-potential of test cases were gathered through the use of program mutation. Empirical results obtained on their approach, however, showed that the method of estimation used in that case did not significantly increase the power of prioritization techniques in comparison to techniques not using the estimation. Moreover, the estimation process itself was expensive. To simplify the estimation process, we predict probabilities using existing static metrics, as shown in Section 2.5. Furthermore, our work investigates the probability that program units contain faults, whereas Elbaum et al. investigate the fault detection probability of test cases if faults do exist, without considering the probability of such existence. In addition to these simplified metrics, in the future we will investigate other metrics to estimate the probability of detecting faults.

Second, our unified approach provides a set of test case prioritization techniques based on various $f_p$ values, whose effectiveness has been evaluated by the empirical study in this paper. However, in addition to effectiveness, the issue of cost should
also be considered when applying test case prioritization techniques. Intuitively, more effort is required to collect coverage at a fine granularity (e.g., statements) than at a coarse granularity (e.g., methods). Because the unified approach did not differ significantly across these granularity levels, using the approach with method coverage appears to be a less costly alternative. As a further issue related to cost, when using the unified approach with differentiated $f_p$ values, we need to calculate the $f_p$ value for each unit; thus, the unified approach with differentiated $f_p$ values is more costly than the unified approach with uniform $f_p$ values. However, considering both effectiveness and cost, the unified approach with differentiated $f_p$ values is more applicable than the unified approach with uniform $f_p$ values, because the former approach usually outperforms the additional strategy whereas the latter does not.

Third, our unified approach is applicable considering both its effectiveness and efficiency. If $n$ is the number of test cases and $m$ is the number of units, the worst case time cost of the total strategy is $O(mn)$ and the worst case time cost of the additional strategy is $O(mn^2)$, whereas the worst case time cost of the approach, including the basic model and the extended model with unified $f_p$ values or differentiated $f_p$ values, is still $O(mn^2)$. That is, the approach is as efficient as the additional strategy, but both of these are less efficient than the total strategy. Considering effectiveness, the approach is usually comparable to the additional strategy, and both of them outperform the total strategy. Furthermore, sometimes the approach even outperforms the additional strategy. Therefore, considering the tradeoff between effectiveness and efficiency, the unified approach may be a good alternative to the total and the additional strategies.

Fourth, considering cost and effectiveness, our unified approach is more beneficial when using method coverage than when using statement coverage. Consider the object program Barbecue as an example. To apply the unified approach based on method coverage, it is necessary to instrument each method so as to learn whether a method is executed by a test. Because the total number of methods in Barbecue is 411, the approach tracks the execution of these 411 methods. To apply the unified approach based on statement coverage, it is necessary to instrument more than 5000 statements since the total number of lines of code in Barbecue is 5.39K. Based on the number of instrumentation points, the statement coverage based approach is more costly than the method coverage based approach. Furthermore, prior work [Mei et al. 2012] (Table 16 in that reference) also demonstrates that it is more expensive to instrument code at the statement level than at the method level. Thus, the unified approach based on statement coverage is much more costly than the unified approach based on method coverage, even for small programs.

To compare the effectiveness of our unified approach based on statement coverage and method coverage, we conducted a further LSD test on the results for Barbecue using the unified approach in the basic model; the results are shown in Table XVIII. As the table shows, there is no significant difference between the two approaches on that program. That is, the unified approach when using method coverage is as effective as when using statement coverage for Barbecue. Similarly, we conducted further LSD tests on the results of Jaxen and JodaTime, which are larger than Barbecue, in order to learn whether the benefit of method coverage based approach may change for larger objects. Based on the results shown in Table XVIII, the benefits seldom increase for larger objects.

6. RELATED WORK
Since there has been a considerable amount of research focusing on various issues in test case prioritization, we partition and discuss the investigated issues in terms of the following categories.
Table XVIII. Results of LSD Tests on Some Java Programs Comparing Dynamic Method Coverage to Dynamic Statement Coverage in the Basic Model

<table>
<thead>
<tr>
<th>TCG</th>
<th>Object</th>
<th>B95</th>
<th>B90</th>
<th>B85</th>
<th>B80</th>
<th>B75</th>
<th>B70</th>
<th>B65</th>
<th>B60</th>
<th>B55</th>
<th>B50</th>
<th>B45</th>
<th>B40</th>
<th>B35</th>
<th>B30</th>
<th>B25</th>
<th>B20</th>
<th>B15</th>
<th>B10</th>
<th>B05</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Barbecue</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Test-Class</td>
<td>jaxen</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>JodaTime</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>-1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

**Prioritization Strategies.** The total and additional strategies (which are both greedy strategies) are the most widely-used prioritization strategies [Rothermel et al. 1999; Hao et al. 2013]. As neither can always achieve the optimal ordering of test cases [Rothermel et al. 1999], researchers have also investigated various other generic strategies. Li et al. [Li et al. 2007] present the 2-optimal strategy (a greedy algorithm based on the k-optimal algorithm [Lin 1965]), a strategy based on hill-climbing, and a strategy based on genetic programming. Jiang et al. [Jiang et al. 2009] present the adaptive random strategy. According to their empirical results, the additional strategy is more effective than the total strategy on average, but the additional strategy is also more expensive than the total strategy, and other strategies fall between the two in terms of effectiveness. The adaptive random strategy is also less expensive than the additional strategy but more expensive than the total strategy on average. In this article, we investigate strategies with flavors of both the total and additional strategies. Techniques derived from our strategies are typically more effective than or as effective as baseline techniques using the total or additional strategies. Theoretically, our strategies are as expensive as the additional strategy.

**Coverage Criteria.** In principle, test case prioritization can use any test adequacy criterion as the underlying coverage criterion. In fact, many criteria have been investigated in previous research on test case prioritization. The most widely used criteria include basic code-based coverage criteria, such as statement and branch coverage [Rothermel et al. 1999], function coverage [Elbaum et al. 2000; Elbaum et al. 2002], block coverage [Do et al. 2004], modified condition/decision coverage [Jones and Harrold 2001; Fang et al. 2012], method coverage [Do et al. 2004] and statically-estimated method coverage [Zhang et al. 2009b; Mei et al. 2012]. There has also been work (described in Section 5) on incorporating information on the probability of exposing faults into criteria [Elbaum et al. 2002]. There has been work [Korel et al. 2006] on test case prioritization using coverage of system models, which can be acquired before the coding phase. Mei et al. [Mei et al. 2009] investigate criteria based on dataflow coverage [Mei et al. 2008] for testing service-oriented software. In this article, we investigate a unified approach that can work with any coverage criterion.

**Constraints.** In practice, there are many constraints affecting test case prioritization. Elbaum et al. [Elbaum et al. 2001] and Park et al. [Park et al. 2008] investigate the constraints of test cost and fault severity. Hou et al. [Hou et al. 2008] investigate the quota constraint on test case prioritization. Kim and Porter [Kim and Porter 2002] investigate resource constraints that may not allow the execution of the entire test suite. Walcott et al. [Walcott et al. 2006] and Zhang et al. [Zhang et al. 2009] investigate time constraints that require the selection of a subset of test cases for prioritization. Do et al. [Do et al. 2008] investigate the use of techniques not specific to time constraints in the presence of those constraints. For constraints that impact only the selection of test cases, our approach may also be applicable.

**Measures.** There has also been research on measuring the effectiveness of test case prioritization techniques. Average Percentage Faults Detected (APFD) [Rothermel et al. 1999] is a widely used metric for assessing test case prior-
itization. Elbaum et al. [Elbaum et al. 2000] propose to calculate APFD relative to test execution time instead of numbers of test cases. Qu et al. [Qu et al. 2007a] propose a normalized APFD to deal with the comparison of prioritized test suites with different numbers of test cases or different execution times. In addition to these general APFD metrics, there are also APFD metrics specific to techniques dealing with constraints. To evaluate techniques dealing with test costs and fault severity, Elbaum et al. [Elbaum et al. 2001; Malishevsky et al. 2006] propose a weighted APFD to assign a higher weight to a test case with lower cost and/or capable of detecting faults of higher severity. To handle issues resulting from missing faults, Qu et al. [Qu et al. 2007b] propose an APFD metric, NAPFD, that normalizes the APFD metric based on the total number of faults detected by the entire test suite. To handle time constraints, Walcott et al. [Walcott et al. 2006] propose an APFD metric with a time penalty, that attempts to favor test cases with less execution time. Finally, Do and Rothermel [Do and Rothermel 2008; Do and Rothermel 2006a] present more comprehensive economic models that can be used to more accurately assess tradeoffs between techniques in industrial testing environments.

Usage Scenarios. Prioritized regression test cases can be used for either a specific subsequent version or a number of subsequent versions. Elbaum et al. [Elbaum et al. 2000; Elbaum et al. 2002] refer to the former as version-specific prioritization and the latter as general prioritization. Although the majority of research on test case prioritization focuses on techniques for general prioritization, some researchers (such as Srivastava and Thiagarajan [Srivastava and Thiagarajan 2002]) have investigated techniques for version-specific prioritization. There is also research (e.g., Elbaum et al. [Elbaum et al. 2000; Elbaum et al. 2002]) investigating the use of general test case prioritization techniques in version-specific prioritization. Like other general prioritization techniques, our unified test case prioritization approach may also be applicable in version-specific prioritization. Furthermore, it should be possible to develop a version-specific technique similar to Srivastava and Thiagarajan’s technique by using our approach on coverage of changed code instead of all code.

Our Prior Conference Publication. This article is a revised and extended version of a paper that appeared in the Proceedings of the International Conference on Software Engineering [Zhang et al. 2013]. This article differs from the conference paper in several ways. First, this article generalizes the prior work from dynamic coverage to static coverage. In particular, the prior paper presents a test-case prioritization approach based solely on dynamic coverage, whereas in this article we provide a unified test-case prioritization approach based on either static coverage or dynamic coverage. Because the static coverage based approach does not require the collection of dynamic coverage information, it is less costly and easier to apply than the dynamic approach. Second, this article reports the results of an empirical study on 40 C objects and 28 Java objects to evaluate the effectiveness of the proposed approach, considering the influence of three internal factors (i.e., model type, $f_p$ values, and coverage type) and three external factors (i.e., coverage granularity, test case granularity, and programming language). This is a much broader study in terms of factors considered than the study presented in the conference paper. Moreover, the conference paper considered only 19 Java objects, whereas this article considers 9 additional Java objects and 40 additional C objects.

7. CONCLUSION
In this article, we have shown how the total and additional test case prioritization strategies can be seen as two extreme instances in models of prioritization strategies. We have shown that there is a spectrum of test case prioritization techniques generated by our models that reside between techniques using purely total and purely
additional strategies. Furthermore, we have proposed extensions to enable the use of differentiated probabilities that test cases can detect faults ($f_p$ values) for methods and the use of static coverage information besides dynamic coverage information.

Our empirical results demonstrate that wide ranges of techniques derived using our basic and extended models can be more effective than total techniques and competitive with additional techniques. Furthermore, our unified approach with differentiated $f_p$ values was more effective than the additional techniques. We studied three internal factors and three external factors, investigating their impact on the effectiveness of test case prioritization, and thus derived the following observations. First, techniques derived from the basic model were less effective than techniques derived from the extended model. Second, techniques with differentiated $f_p$ values were more effective than techniques with uniform $f_p$ values. Third, techniques based on dynamic coverage were slightly more effective than techniques based on static coverage, and these techniques sometimes exhibited no significant differences. Fourth, techniques using method coverage were more beneficial than techniques using statement coverage. Fifth, our unified approach was more effective when applied to test cases at the test-method level than at the test-class level. Finally, our unified approach was more effective when applied to Java programs with unit tests than when applied to C programs with system tests.

In a broader sense, we view our results as a fundamental step toward controlling the uncertainties of fault detection in test case prioritization. In this sense, our models provide a new dimension for creating better prioritization techniques.

ACKNOWLEDGMENTS

This work was supported in part by the National 973 Program of China No. 2014CB347701, the High-Tech Research and Development Program of China under Grant No.2013AA01A605, the Science Fund for Creative Research Groups of China No. 61121063, the National Science Foundation of China No. U1201252, 61228203, 61225007, and 61272157, the National Science Foundation through award CNS-0720757, and the Air Force Office of Scientific Research through award FA9550-10-1-0406. We thank Sarfraz Khurshid for his insightful comments and suggestions.

REFERENCES


A Unified Test Case Prioritization Approach


Received February 2014; revised March 2014; accepted September 2014

ACM Transactions on Software Engineering and Methodology, Vol. 9, No. 4, Article 39, Publication date: March 2010.