Bridging the Gap between the *Total* and *Additional* Test-Case Prioritization Strategies

Lingming Zhang[*†], Dan Hao[*], Lu Zhang[*], Gregg Rothermel[‡], Hong Mei[*]
[*]Key Laboratory of High Confidence Software Technologies (Peking University), MoE, Beijing, 100871, China
{zhanglm07,haod,zhanglu,meih}@sei.pku.edu.cn
[†]Department of Electrical and Computer Engineering, University of Texas, Austin, 78712, USA
zhanglm@utexas.edu
[‡]Department of Computer Science and Engineering, University of Nebraska, Lincoln, 68588, USA
grother@cse.unl.edu

***Abstract*—In recent years, researchers have intensively investigated various topics in test-case prioritization, which aims to re-order test cases to increase the rate of fault detection during regression testing. The *total* and *additional* prioritization strategies, which prioritize based on total numbers of elements covered per test, and numbers of additional (not-yet-covered) elements covered per test, are two widely-adopted generic strategies used for such prioritization. This paper proposes a basic model and an extended model that unify the *total* strategy and the *additional* strategy. Our models yield a spectrum of generic strategies ranging between the *total* and *additional* strategies, depending on a parameter referred to as the *p* value. We also propose four heuristics to obtain differentiated *p* values for different methods under test. We performed an empirical study on 19 versions of four Java programs to explore our results. Our results demonstrate that wide ranges of strategies in our basic and extended models with uniform *p* values can significantly outperform both the *total* and *additional* strategies. In addition, our results also demonstrate that using differentiated *p* values for both the basic and extended models with method coverage can even outperform the *additional* strategy using statement coverage.**

I. Introduction

Software engineers usually maintain a large number of test cases, which can be reused in regression testing to test software changes. Due to the large number of test cases, regression testing can be very time consuming. For example, Elbaum et al. [7], [9] reported one industrial case in which the execution time of the entire regression test suite for one product was seven weeks. Test-case prioritization [7]–[9], [24], [26], [30], which attempts to re-order regression test cases to detect faults as early as possible, has been intensively investigated as a way to deal with lengthy regression testing cycles.

In test-case prioritization, a fundamental topic involves prioritization strategies. In previous work, researchers have studied two greedy strategies (the *total* and *additional* strategies), which are generic for different coverage criteria. Given a coverage criterion, the *total* strategy sorts test cases in descending order of coverage, whereas the *additional* strategy always picks a next test case having the maximal coverage of items not yet covered by previously prioritized test cases. In addition to these two strategies, researchers have also investigated other generic strategies. Li et al. [16] investigated the 2-optimal greedy strategy [17], a hill-climbing strategy, and a genetic programming strategy. Jiang et al. [12] investigated adaptive random prioritization. Their empirical results show that the *additional* strategy remains the most effective generic strategy on average in terms of rate of fault detection.

There is also, however, a weakness in the *additional* strategy. Consider statement coverage for instance. In the *additional* strategy, after a test case t is chosen, no statement covered by t is explicitly considered again until all coverable statements are covered at least one time. As a result, when there is a fault f in one statement covered by t but not covered by any test case chosen before t, if t cannot detect f, the detection of f may be greatly postponed. In contrast, the *total* strategy does not have this weakness, because when choosing a next test case, the *total* strategy always considers all statements no matter whether or not previously chosen test cases have covered the statements. This said, as the *total* strategy counts only the numbers of statements covered by each test case, it may be more inclined to choose test cases to cover statements previously covered many times than to choose test cases to cover previously not (intensively) covered statements. Thus, the *total* strategy may postpone the detection of faults in rarely covered statements. As a result, some strategy that has the flavor of both the *additional* strategy and the *total* strategy may be more advantageous.

To understand the situation in which a test case covers a statement but does not reveal a fault in the statement, consider the following piece of code with a fault in line 5. The code is a method returning the larger of x and y. A test case in which the value of x is smaller than that of y covers the faulty statement and detects the fault. However, a test case in which the value of x is equal to that of y also covers the faulty statement but does not detect the fault.

```
1:    int max(int x, int y) {
2:        if (x>y)
3:            return x;
4:        else
5:            return x;//should be "return y".
6:    }
```

In fact, research on test-suite reduction [11], [25], [31] has demonstrated that re-covering already covered statements may enhance fault-detection capability. Furthermore, when we

978-1-4673-3076-3/13 © 2013 IEEE

ICSE 2013, San Francisco, CA, USA

Accepted for publication by IEEE. © 2013 IEEE. Personal use of this material is permitted. Permission from IEEE must be obtained for all other uses, in any current or future media, including reprinting/republishing this material for advertising or promotional purposes, creating new collective works, for resale or redistribution to servers or lists, or reuse of any copyrighted component of this work in other works.

consider test-case prioritization based on coverage information at a coarser level (e.g., the method level), it may be more common for a test case to miss a fault in a method covered by the test case, because that test case may fail to cover the faulty statement in the method.

In this paper, we propose a unified view (including a basic model and an extended model) for generic strategies in test-case prioritization. In our models, the *total* and *additional* strategies are extreme instances, and the models also define various generic strategies that lie between the *total* strategy and the *additional* strategy depending on the value of fault detection probability (referred to as the *p* value). In addition, we further extend the models by using differentiated *p* values. We view our models as an initial framework to control the uncertainty of fault detection during test-case prioritization, and believe more techniques can be derived based on our models. We performed an empirical study to compare our strategies with the *total* strategy and the *additional* strategy. Our results demonstrate that many of our strategies can outperform both the *total* and *additional* strategies.

The main contributions of this paper are as follows.

- A new approach that creates better prioritization techniques by controlling the uncertainty of fault detection capability in test-case prioritization.
- Two models that unify the *total* and *additional* strategies and can also yield a spectrum of generic strategies having flavors of both the *total* and *additional* strategies.
- Empirical evidence that many strategies between the *total* and *additional* strategies are more effective than either of those strategies.
- Empirical evidence that our strategies using differentiated *p* values with method coverage can significantly outperform the *additional* strategy with statement coverage.

II. Unifying the Total and Additional Strategies

With the *additional* strategy, the primary concern is to cover units not yet covered by previous test cases. This strategy should be well suited for circumstances in which the probability of a test case detecting faults in units it covers is high. On the other hand, the primary concern for the *total* strategy is to cover the most units with each test case. This strategy should be well suited for circumstances in which the probability of a test case detecting faults in units it covers is low. Thus, if we explicitly consider the probability for a test case to detect faults in units it covers, we may devise strategies to take advantage of the strengths of both the *total* and *additional* strategies. More specifically, our models initially assign probability values for each program $unit$[1]. Then, each time a $unit$ is covered by a test case (that could potentially detect some fault(s) in the $unit$), the probability that the $unit$ contains undetected faults is reduced by some ratio between 0% (as in the *total* strategy) and 100% (as in the *additional* strategy). In this way, we build a spectrum of generic prioritization strategies between the *total* and *additional* strategies.

[1] As our approach is intended to work with different coverage criteria, we use $unit$ as a generic term to denote different structural elements used in different coverage criteria. For example, a $unit$ represents a $statement$ for statement coverage but a $method$ for method coverage.

Algorithm 1 Prioritization in the basic model with p

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

A. Basic Model

In our basic model, when a test case t covers a unit u, we refer to the probability that t can detect faults in u as p. Consider a test suite $T = \{t_1, t_2, ..., t_n\}$ containing n test cases and a program $U = \{u_1, u_2, ..., u_m\}$ containing m units. Algorithm 1 depicts test-case prioritization in our basic model, in which we use a Boolean array $Cover[i, j]$ $(1 \leq i \leq n, 1 \leq j \leq m)$ to denote whether test case t_i covers unit u_j.

In Algorithm 1, we use an array $Prob[j]$ $(1 \leq j \leq m)$ to store the probability that unit u_j contains undetected faults. Initially, we set the value of $Prob[j]$ $(1 \leq j \leq m)$ to be 1. We use a Boolean array $Selected[i]$ $(1 \leq i \leq n)$ to

store whether test case t_i has been selected for prioritization. Initially, we set the value of $Selected[i]$ $(1 \leq i \leq n)$ to be $false$. Furthermore, we use an 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 ith position.

In Algorithm 1, lines 1-6 perform initialization. In the main loop from lines 7 to 39, each iteration determines which test case to place in the prioritized test suite. Lines 8-31 find a test case t_k such that t_k is previously not chosen and the sum of probabilities that units covered by t_k contain undetected faults is the highest among test cases not yet chosen. Note that, as the basic model utilizes a *uniform* probability 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 check whether there is another unchosen test case t_l for which the sum of the probabilities that the covered units contain undetected faults is higher than that for t_k. Line 32 sets the ith 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 is in fact a variant of the algorithm for the *additional* strategy. The main difference is that this algorithm tries to find the test case covering units with the maximal sum of probabilities of containing undetected faults. Due to the similarity between this algorithm and the *additional* strategy, its worst case time cost is the same as that of the *additional* strategy (i.e., $O(mn^2)$, where n is the number of test cases and m is the number of units [26]).

With Algorithm 1, an optimistic tester who believes that a test case is likely to detect faults in covered units may set the value of p to 1. In such a circumstance, this algorithm is equivalent to the *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 units covered by the test case may set the value of p to 0. In such a circumstance, this algorithm is equivalent to the *total* strategy. The reason is that lines 34-38 do not change the probability that any previously covered unit contains undetected faults. Note that setting p to 0 does not render the algorithm as efficient as the original *total* strategy, whose worst case time cost is $O(mn)$ [26]. Finally, if a tester sets the value of p to a number between 0 and 1, this algorithm is a strategy between the *total* strategy and the *additional* strategy. The closer p is to 0, the closer this algorithm is to the *total* strategy; and the closer p is to 1, the closer this algorithm is to the *additional* strategy.

B. Extended Model

In our basic model and previously proposed strategies for test-case prioritization, when a test case t covers a unit u, the number of times t covers u is not further considered. That is, no matter how many times t covers u, the algorithm treats u as having been covered once. Intuitively, the more times t covers u, the more probable it may be for t to detect faults in u. Therefore, considering the ability of a test case to cover a unit multiple times may result in higher effectiveness.

We now 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 in our basic model, but the extended algorithm uses a different method for calculating the probability for a test case to detect previously undetected faults. We extend $Cover[i, j]$ $(1 \leq i \leq n,\ 1 \leq j \leq m)$ to denote the number of times test case t_i covers unit u_j. We now present the main differences between the two algorithms.

First, as the number of times test case t_k covers unit u_j is $Cover[k, j]$, the probability for unit u_j to contain undetected faults changes from $Prob[j]$ to $Prob[j] * (1-p)^{Cover[k,j]}$ after executing t_k if we consider each instance of coverage to have an equal probability p of detecting faults. That is to say, for unit u_j alone, execution of t_k decreases the probability that u_j contains undetected faults by $Prob[j] * (1-(1-p)^{Cover[k,j]})$. Thus, in the extended algorithm, we change lines 15 and 23 of Algorithm 1 to $sum \leftarrow sum + Prob[j] * (1-(1-p)^{Cover[k,j]})$ and $s \leftarrow s + Prob[j] * (1 - (1 - p)^{Cover[l,j]})$, respectively.

Second, after we select test case t_k for prioritization at the ith place, the probability for unit u_j to contain undetected faults changes from $Prob[j]$ to $Prob[j] * (1 - p)^{Cover[k,j]}$. Thus, in the extended algorithm, we change line 36 of Algorithm 1 to $Prob[j] \leftarrow Prob[j] * (1 - p)^{Cover[k,j]}$. The worst case time cost of the extended algorithm is also $O(mn^2)$, the same as that of Algorithm 1.

In the extended algorithm, if we set p to 1, the algorithm is the same as the *additional* strategy, because $(1 - p)^{Cover[k,j]}$ is equal to 0 when p is equal to 1. However, if we set p to 0, the extended algorithm cannot distinguish any test cases from each other,[2] because $1-(1-p)^{Cover[k,j]}$ is always equal to 0 when p is equal to 0. If we set 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.

C. Differentiating p Values

In Section II-A, in our basic model, whenever a test case t covers a unit u, we consider the probability for t to detect faults in u to be uniformly p. In Section II-B, using our extended model, we reason that when t covers u multiple times, the probability for t to detect faults in u may not be uniform, but each instance of coverage also implies a uniform probability of fault detection. In reality, however, faults in some units may be easier to detect than faults in other units.

In this section, we further extend our models to account for the situation in which the probability of fault detection is differentiated. To deal with this situation, we need to

[2]This limitation is due to the specific algorithm, but conceptually our extended model implementation yields the *total* strategy when $p = 0$.

assign different probability values for test cases to detect faults in different units. The challenge in performing such an assignment, however, lies in obtaining effective estimates of the probability of fault detection. In this paper, we further estimate differentiated 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 Cyclomatic Complexity [19]. 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 the faults within them, i.e., they should have lower p values. We believe that test cases should be good at detecting faults, and thus we calculate the p value for each method in the range $[0.5, 1.0]$. Formally, we use both linear normalization (Formula (1)) and log normalization (Formula (2)) to calculate the p value for the jth method (i.e., $p[j]$) as follows:

$$1 - 0.5 * \frac{Metric[j] - Metric_{min}}{Metric_{max} - Metric_{min}} \quad (1)$$

$$1 - 0.5 * \frac{log_{10}(Metric[j] + 1) - log_{10}(Metric_{min} + 1)}{log_{10}(Metric_{max} + 1) - log_{10}(Metric_{min} + 1)} \quad (2)$$

where $Metric[j]$ denotes the MLoC or McCabe metric values for the jth method, and $Metric_{min}$/$Metric_{max}$ denotes the minimum/maximum metric value among all methods of the program under test.[3]

Based on the two metrics and the two p calculation formulas, we thus have four heuristics for generating a differentiated p value for each method. For both models, we change all references to the uniform p into the differentiated $p[j]$ generated for the specific jth method. For the basic model, we change line 36 of Algorithm 1 into $Prob[j] \leftarrow Prob[j] * (1 - p[j])$. Similarly, for the extended model, we change lines 15, 23, and 36 of Algorithm 1 to $sum \leftarrow sum + Prob[j] * (1 - (1 - p[j])^{Cover[k,j]})$, $s \leftarrow s + Prob[j] * (1 - (1 - p[j])^{Cover[l,j]})$, and $Prob[j] \leftarrow Prob[j] * (1 - p[j])^{Cover[k,j]}$, respectively. Note that the worst case time costs of the basic and extended models with differentiated p values are still $O(mn^2)$.

III. Empirical Study

To evaluate our strategies with uniform and differentiated p values in the basic and extended models, we performed an empirical study to investigate the following research questions:

- **RQ1**: How do prioritization strategies generated by the basic and extended models with uniform p values compare with the *total* and *additional* strategies?
- **RQ2**: How do the granularity of coverage and the granularity of test cases impact the comparative effectiveness of strategies generated by our models?
- **RQ3**: How does the use of differentiated p values compare, in terms of effectiveness, with the *total* and *additional* strategies?

[3]Note that all metric values are increased by 1 in the log normalization to avoid the $log_{10}0$ exception.

A. Independent Variables

We consider three independent variables:

Prioritization Strategy. We use the following 48 strategies for test-case prioritization. First, as control strategies, we use the *total* and *additional* strategies. Second, for our basic model we use values of p ranging from 0.05 to 0.95 with increments of 0.05, i.e., 19 p values. Third, for our extended model we use the same 19 values of p as those used for our basic model. Fourth, for differentiated p values we use the four p value generation heuristics for both the basic and extended models.

Coverage Granularity. In prior research on test-case prioritization, researchers treated coverage granularity as a constituent part of prioritization techniques. As our aim is to investigate various generic prioritization strategies, we separate coverage granularity from prioritization techniques. As in prior research, we use structural coverage criteria at both the method level and the statement level. Note that we used differentiated p values only at the method level.

Test-Case Granularity. We consider test-case granularity as an additional factor, at two levels: the test-class level and the test-method level. For the test-class level we treat each JUnit TestCase class as a test case. For the test-method level we treat each test method in a JUnit TestCase class as a test case. That is to say, a test case at the test-class level typically consists of a number of test cases at the test-method level. Section III-C provides a detailed description.

B. Dependent Variable

Our dependent variable tracks technique effectiveness. We adopt the well-known APFD (Average Percentage Faults Detected) metric [26]. Let T be a test suite and T' be a permutation of T, the APFD for T' is defined as follows.

$$APFD = \frac{\sum_{i=1}^{n-1} F_i}{n * l} + \frac{1}{2n} \quad (3)$$

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'.

C. Object Programs, Test Suites, and Faults

As objects of study we consider 19 versions of four programs written in Java, including three versions of *jtopas*, three versions of *xml-security*, five versions of *jmeter*, and eight versions of *ant*. We obtained the programs from the *Software-artifact Infrastructure Repository* (SIR)[4] [3], which provides Java and C programs for controlled experimentation on program analysis and testing. The sizes of the programs range from 1.8 to 80 KLoC. Table I depicts statistics on the objects. In Table I, for each object program, Columns 3 and 4 present the number of classes (including interfaces) and the number of methods, respectively.

In SIR, each version of each program has a JUnit test suite that was developed during program evolution. Due to the features of JUnit, there are two levels of test-case granularity in

[4]http://sir.unl.edu/portal/index.html, accessed in February 2013.

TABLE I
STATISTICS ON OBJECTS OF STUDY

Object	KLoC	#Class	#Meth	#TClass	#TMeth
$jtopas$-$v1$	1.89	19	284	10 (8)	126 (24)
$jtopas$-$v2$	2.03	21	302	11 (10)	128 (27)
$jtopas$-$v3$	5.36	50	748	18 (8)	209 (25)
$xmlsec$-$v1$	18.3	179	1627	15 (3)	92 (10)
$xmlsec$-$v2$	19.0	180	1629	15 (1)	94 (7)
$xmlsec$-$v3$	16.9	145	1398	13 (8)	84 (41)
$jmeter$-$v1$	33.7	334	2919	26 (7)	78 (18)
$jmeter$-$v2$	33.1	319	2838	29 (8)	80 (31)
$jmeter$-$v3$	37.3	373	3445	33 (16)	78 (43)
$jmeter$-$v4$	38.4	380	3536	33 (16)	78 (55)
$jmeter$-$v5$	41.1	389	3613	37 (20)	97 (57)
ant-$v1$	25.8	228	2511	34 (17)	137 (45)
ant-$v2$	39.7	342	3836	51 (42)	219 (118)
ant-$v3$	39.8	342	3845	51 (44)	219 (148)
ant-$v4$	61.9	532	5684	102 (47)	521 (135)
ant-$v5$	63.5	536	5802	105 (53)	557 (133)
ant-$v6$	63.6	536	5808	105 (52)	559 (230)
ant-$v7$	80.4	649	7520	149 (122)	877 (599)
ant-$v8$	80.4	650	7524	149 (51)	878 (197)

these test suites: the test-class level and the test-method level. Column 5 of Table I depicts the number of all test cases and the number of test cases that detect at least one studied fault for each program at the test-class level. Similarly, Column 6 depicts the test case statistics at the test-method level. As previous research [1], [2], [5] has confirmed that it is suitable to use faults produced via mutation for experimentation in test-case prioritization, we followed a similar procedure to produce faulty versions for each of the 19 object programs. In particular, we used *MuJava*[5] [18] to generate faults and followed the procedure used by Do et al. [5] to select specific mutants to use (as detailed below).

D. Implementation

To collect coverage information, 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*.[6] To compute Java metrics for each method, we implemented our tool based on the abstract syntax tree (AST) analysis provided by the *Eclipse JDT toolkit*.[7] We extended the Eclipse AST parsing tool to calculate method lines of code (MLoC) and McCabe Cyclomatic complexity (McCabe) metrics.

E. Experiment Procedure

In actual testing scenarios, a specific program version usually does not contain a large number of faults [5]. Therefore, similar to Do et al. [5], we used the mutant pool for each object program to create a set of small mutant groups. To form a mutant group, we randomly selected five mutants that can be killed by one or more test cases in the test suite for the program. For each program, we randomly produced up to 20 mutant groups for each program ensuring that no mutant is used in more than one mutant group. In fact, as there are only 35 mutants of *jmeter-v1* that can be killed by one or more test cases in its test suite, we produced only seven mutant groups for this program. In all other circumstances we produced 20 mutant groups for each program.

Next, we used each of the mutant groups produced for each of the 19 program versions as possible subsequent versions. That is to say, given a program version V and a generic prioritization strategy S with a coverage-granularity level C_l and a test-case-granularity level T_l, we obtained the effectiveness of strategy S on V for C_l and T_l as follows. First, we used S to obtain a prioritized sequence of test cases for V at C_l and T_l. Then, we calculated the APFD values of the prioritized sequence of test cases for each mutant group of V. These values serve as our data sets for analysis.

F. Threats to Validity

Our object programs, test cases, and seeded faults may all pose threats to external validity. First, although we used 19 Java program versions of various sizes, the differences seen in our study may be difficult to generalize to other Java programs. Furthermore, our results may not generalize to programs written in languages other than Java. Second, our results based on programs with seeded faults may not be generalizable to programs with real faults. Third, the results may not be generalizable to other test cases. Further reduction of these threats requires additional studies involving additional object programs, test suites, and faults.

The main threat to internal validity for our study is that there may be faults in our implementation of the strategies and the calculation of APFD values. To reduce this threat, we reviewed all the code that we produced for our experiments before conducting the experiments.

To assess technique effectiveness, we used the APFD metric [26] that is widely used for test-case prioritization. However, the APFD metric does have limitations [5], [26], and we did not consider efficiency or other cost and savings factors. Reducing this threat requires additional studies using more sophisticated cost-benefit models [8].

G. Results and Analysis

Due to the large number of strategies, various test-case granularities, coverage granularities, objects, and mutant groups studied, box-plots across all objects are the most suitable way to show our results.

1) RQ1: Existence of Better Strategies Between the Total and Additional Strategies: Figures 1 to 4 depict the results of the comparision of the 19 strategies in our basic model and the 19 strategies in our extended model with the *total* and *additional* strategies using test suites at the test-method/test-class level and coverage information at the method/statement level. We denote the *total* strategy as *Tot.* and the *additional* strategy as *Add.*. For a strategy in our basic model, we use B and the value of p to denote the strategy. For example, we use $B05$ to denote the strategy in our basic model with the p value 0.05. Similarly, for a strategy in our extended model,

[5]http://cs.gmu.edu/~offutt/mujava, accessed in February 2013.
[6]http://asm.ow2.org, accessed in February 2013.
[7]http://www.eclipse.org/jdt/, accessed in February 2013.

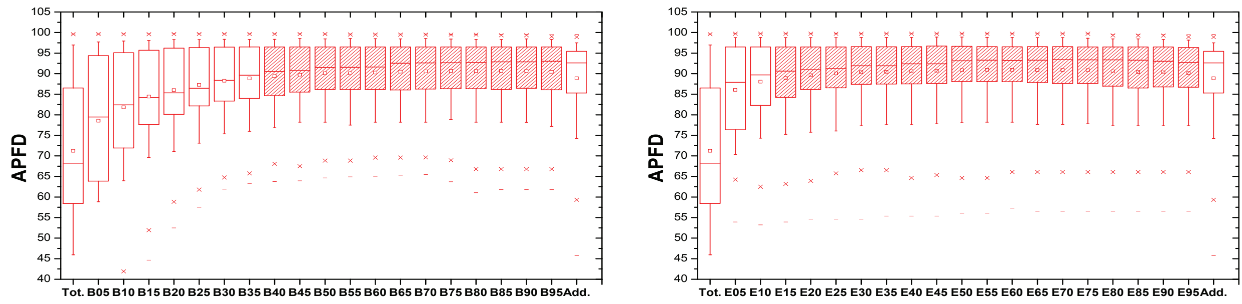


Fig. 1. Results for test suites at the test-method level with method coverage

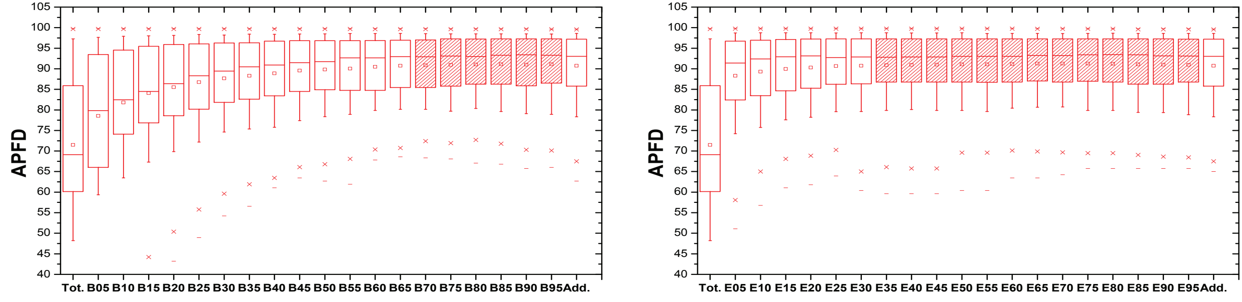


Fig. 2. Results for test suites at the test-method level with statement coverage

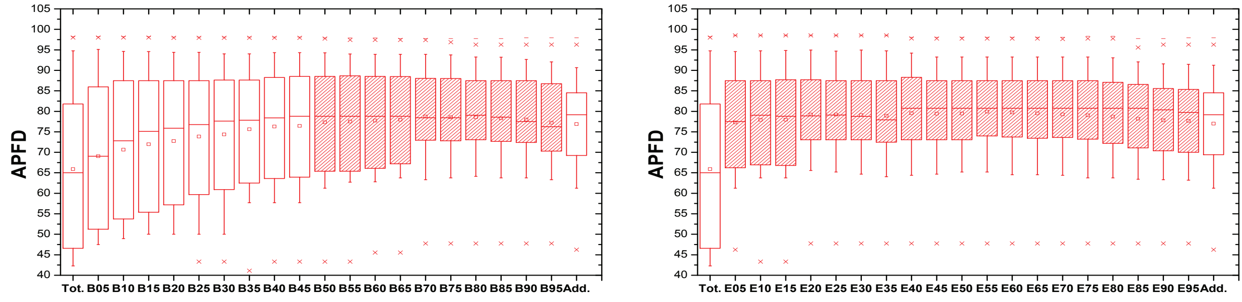


Fig. 3. Results for test suites at the test-class level with method coverage

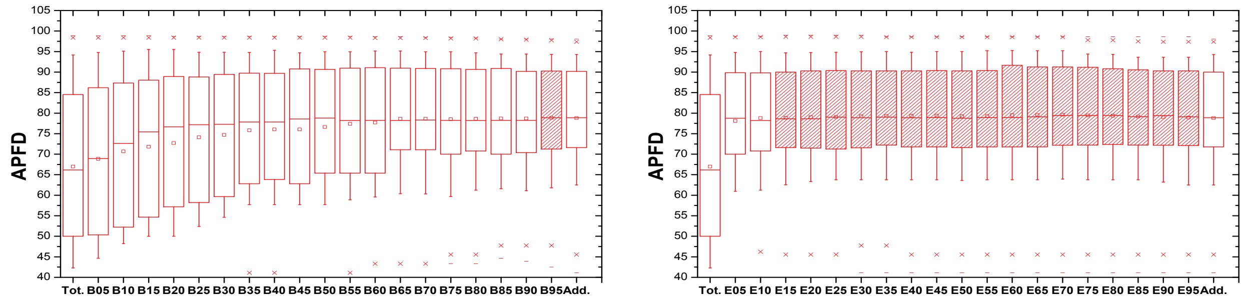


Fig. 4. Results for test suites at the test-class level with statement coverage

we use E and the value of p to denote the strategy. Thus, the strategy in our extended model with the value of p set to 0.05 is denoted as $E05$. In each plot, the X-axis shows various strategies compared, and the Y-axis shows the APFD values measured. Each box plot shows the average (dot in the box), median (line in the box), upper/lower quartile, and 90th/10th percentile APFD values achieved by a strategy over all mutant groups of all 19 versions. For ease of understanding, we mark the strategies with higher average APFD values over the corresponding *additional* strategies as shadowed box plots. Based on the results, we make the following observations.

First, when comparing strategies in our approach with the *additional* strategy, strategies with p values between 0.95 and 0.50 in both our basic and extended models typically achieve higher average APFD values. The only exceptions to this are the strategies in our basic model based on statement coverage for test suites at the test-class level with p values between 0.90 and 0.50, and for test suites at the test-method level with p values between 0.65 and 0.50. This observation indicates that there is a wide range of p values that can be used for our models. It should also be noted that the average increases in APFD of our strategies over the *additional* strategy are usually not large. However, considering that the *additional* strategy is widely accepted as the most effective prioritization strategy and is as expensive as our strategies, the increases in APFD are valuable and are actually achieved with almost no extra cost.

TABLE II

FISHER'S LSD TEST FOR COMPARING STRATEGIES IN THE BASIC MODEL TO THE *additional* STRATEGY

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

TABLE III

FISHER'S LSD TEST FOR COMPARING STRATEGIES IN THE EXTENDED MODEL TO THE *additional* STRATEGY

TCG	CG	$E95$	$E90$	$E85$	$E80$	$E75$	$E70$	$E65$	$E60$	$E55$	$E50$	$E45$	$E40$	$E35$	$E30$	$E25$	$E20$	$E15$	$E10$	$E05$
Test-Method	Method	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0	0	0	0	-1
	Statement	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	-1	-1
Test-Class	Method	0	0	0	0	1	1	1	1	1	1	1	1	1	1	1	1	0	0	0
	Statement	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Second, when comparing strategies in our approach with the *total* strategy (denoted as *Tot.* in the figures), our strategies with all p values in both our basic and extended models achieve higher average APFD values. One interesting point is that, even when the p value is 0.05 (which results in strategies similar to the *total* strategy), strategies in both our basic and extended models are substantially more effective than the *total* strategy. This observation indicates that adding a little flavor of the *additional* strategy into the *total* strategy could improve the *total* strategy substantially.

Third, when comparing strategies in our basic model and strategies in our extended model, the strategies perform similarly with p values close to 1 and differently with p values close to 0. When p is close to 1, strategies in both models achieve comparable and even higher APFD values than the *additional* strategy. However, when p is close to 0, strategies in our extended model remain competitive but strategies in our basic model become much less competitive. In other words, strategies with a small p value in our basic model perform more like the *total* strategy, but strategies in our extended model always perform like the *additional* strategy with any p values. In fact, almost all strategies of our extended model with $p \geq 0.15$ outperform the *additional* strategies, except those prioritizing tests at the test-method level using statement coverage with $p \in [0.15, 0.30]$.

As strategies in our models and the *additional* strategy typically achieve similar APFD values, for each coverage-granularity level and each test-case-granularity level, we used Origin[8] to perform a one-way ANOVA analysis of the strategies. The results indicate that there are significant differences among the strategies at the 0.05 significance level. We then used Origin to perform Fisher's LSD test [29] of the strategies. Tables II and III (in which TCG stands for test-case granularity, CG stands for coverage granularity, 1 indicates statistically significantly better, 0 indicates no significant difference, and -1 indicates statistically significantly worse) list the results of Fisher's LSD test for comparing strategies in our models to the *additional* strategy.

According to Tables II and III, when prioritizing test cases at the test-method level using method coverage, strategies with p values between 0.65 and 0.95 in our basic model and with any p values between 0.30 and 0.95 in our extended model achieve significantly better APFD values than the *additional* strategy. When prioritizing test cases at the test-class level using method coverage, strategies with p values between 0.20 and 0.75 in our extended model significantly outperform the *additional* strategy. Furthermore, the *additional* strategy cannot significantly outperform any strategies in our basic model with p values between 0.50 and 0.95 and any strategies in our extended model with p values between 0.15 and 0.95 in any circumstance. This observation further confirms that our models can achieve clear benefits.

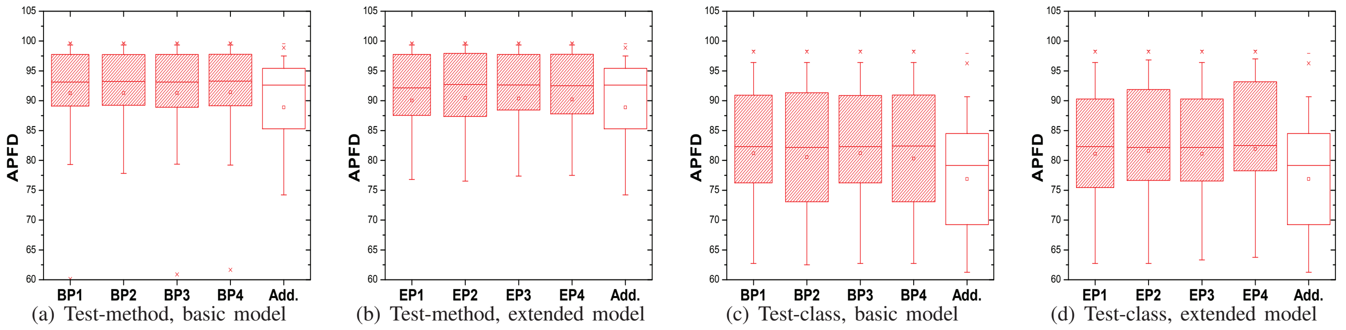
2) RQ2: Impact of Coverage and Test-Case Granularities: Based on Figures 1 to 4, we make the following observations.

Impact of coverage granularity. Our models seem to be more beneficial when using coverage information at the method level than at the statement level. According to comparisons between Figure 1 and Figure 2, and between Figure 3 and Figure 4, in both our basic and extended models, the ranges in which our strategies outperform the *additional* strategy on average are much broader using coverage information at the method level than at the statement level. We suspect the reason for this to be that, when a test case covers a statement, the probability for the test case to detect faults in the statement is very high. Thus, the *additional* strategy is already a good enough strategy for this situation. However, when a test case covers a method, the probability for the test case to detect faults in the covered method is not very high. Thus, we should typically consider that the method may still contain some undetected faults after being covered by some test cases.

Our extended model seems to be applicable for both method coverage and statement coverage. In fact, for all combinations of coverage granularity and test-case granularity, the ranges of strategies in our extended model that outperform the *additional* strategy on average are all very wide (i.e., for any $p > 0.30$).

As our empirical results indicate that our models are more beneficial with method coverage, we further compare our strategies using method coverage with the *additional* strategy using statement coverage. When prioritizing test cases at the test-method level, the average APFD values of wide ranges of strategies in our models (i.e., strategies in the basic model with p values between 0.75 and 0.90, and strategies in the extended model with p values between 0.45 and 0.75) using method coverage are very close to the average APFD values of the *additional* strategy using statement coverage. When prioritizing test cases at the test-class level, the average

[8] http://www.originlab.com/, accessed in February 2013.



(a) Test-method, basic model (b) Test-method, extended model (c) Test-class, basic model (d) Test-class, extended model

Fig. 5. Prioritization results for models embodied with differentiated p values for each method

APFD values of wide ranges of strategies in our models (i.e., strategies in the basic model with p values between 0.70 and 0.80, and strategies in the extended model with p values between 0.20 and 0.80) using method coverage are as competitive as or even better than the average APFD values of the *additional* strategy using statement coverage. We also performed an ANOVA analysis (at the 0.05 level) and Fisher's LSD test to compare our strategies and the *additional* strategy using method coverage to the *additional* strategy using statement coverage. The ANOVA analysis and Fisher's LSD test demonstrate that there is no statistically significant difference between the *additional* strategy using statement coverage and any strategy in our basic model with any p value between 0.50 and 0.95 or any strategy in our extended model with any p value between 0.20 and 0.95 using method coverage. However, the *additional* strategy using statement coverage is significantly better than the *additional* strategy using method coverage. As coverage information at the method level is usually much less expensive to acquire than coverage information at the statement level, this result indicates that wide ranges of strategies in our models using method coverage can serve as cheap alternatives for the *additional* strategy using statement coverage.

Impact of test-case granularity. Our extended model seems to be more beneficial than our basic model for prioritizing test cases at the test-class level. First, when using the extended model instead of the basic model, the number of strategies that outperform the *additional* strategies increases more dramatically at the test-class level than at the test-method level (shown in Figures 1 to 4). Second, when prioritizing test cases at the test-method level, the largest average APFD values achieved by our extended model are larger than those achieved by our basic model by 0.15 (statement coverage) and 0.25 (method coverage), respectively. However, when prioritizing test cases at the test-class level, the differences are 0.69 (statement coverage) and 1.03 (method coverage), respectively. Third, results of our statistical analysis shown in Tables II and III also confirm this observation. We suspect the reason for this to be that it is more common for a test case at the test-class level than a test case at the test-method level to cover a method or a statement more than once. In such a circumstance, it is more beneficial to consider multiple coverage information.

All the strategies that we considered achieve significantly higher average APFD values for prioritizing test cases at the test-method level than for 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 for prioritizing test cases at the test-method level is uniformly higher than that for prioritizing test cases at the test-class level. We suspect the reason for this to be that, as 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.

TABLE IV

FISHER'S LSD TEST FOR COMPARING p-DIFFERENTIATED TECHNIQUES WITH p-UNIFORM TECHNIQUES AT THE TEST-METHOD LEVEL

Tech.	*BP*1	*BP*2	*BP*3	*BP*4	*EP*1	*EP*2	*EP*3	*EP*4
M-B75	0	0	0	0	0	0	0	0
M-E60	0	0	0	0	0	0	0	0
M-*Add.*	1	1	1	1	1	1	1	1
S-B85	0	0	0	0	0	0	0	0
S-E65	0	0	0	0	0	0	0	0
S-*Add.*	0	0	0	0	0	0	0	0

TABLE V

FISHER'S LSD TEST FOR COMPARING p-DIFFERENTIATED TECHNIQUES WITH p-UNIFORM TECHNIQUES AT THE TEST-CLASS LEVEL

Tech.	*BP*1	*BP*2	*BP*3	*BP*4	*EP*1	*EP*2	*EP*3	*EP*4
M-B70	1	0	1	0	1	1	1	1
M-E55	0	0	0	0	0	0	0	1
M-*Add.*	1	1	1	1	1	1	1	1
S-B95	1	0	1	0	1	1	1	1
S-E70	0	0	0	0	0	1	0	1
S-*Add.*	1	0	1	0	1	1	1	1

3) RQ3: Using Differentiated p Values: Figure 5 depicts results obtained by comparing the p-differentiated strategies with the corresponding *additional* strategies. We use BP to denote the four strategies in the basic model, and EP to denote the four strategies in the extended model. For the basic model, *BP*1 denotes the use of the MLoC metric and linear normalization, *BP*2 denotes the use of the MLoC metric and log normalization, *BP*3 denotes the use of the McCabe metric and linear normalization, and *BP*4 denotes the use of the McCabe metric and log normalization. The naming of strategies in the extended model follows the same manner. In the box plots, the X-axis denotes the studied strategies, the Y-axis denotes the APFD values achieved by compared strategies, and each box denotes the results of a strategy on all mutant groups of all objects. We also performed an ANOVA analysis (at the 0.05 level) and Fisher's LSD test to compare the eight strategies with differentiated p values to the best strategies in our basic/extended models and *additional*

strategies using method and statement coverage. Tables IV and V show the Fisher's LSD test result, where "M-" denotes the strategies using method coverage, and "S-" denotes the strategies using statement coverage. For example, "M-$B70$" denote the $B70$ strategy using method coverage. We make the following observations.

First, all strategies with differentiated p values outperform the corresponding *additional* strategies based on method coverage substantially. Figure 5 shows that all the strategies with differentiated p values achieve higher APFD values over corresponding addtional strategies on average. For example, when prioritizing test-class-level tests using method coverage, the *additional* strategy achieves an APFD value of 76.88 on average, while the four strategies from the extended model achieve APFD values from 81.10 to 81.92. In addition, Table IV shows that all eight strategies are statistically significantly better than the *additional* strategy based on method coverage under test-method granularity, and Table V shows that all eight strategies are statistically significantly better than the *additional* strategy based on method coverage under test-class granularity.

Second, all strategies with differentiated p values using method coverage are comparable to the best strategies in our basic and extended models (including strategies using method and statement coverage) and the *additional* strategies using statement coverage, and even outperform some of those techniques. At both test-class and test-method granularities, the eight strategies are not statistically inferior to any best strategies within our basic/extended models or *additional* strategies using statement coverage. At the test-class granularity, six of the eight strategies are statistically significantly better than the *additional* strategy using statement coverage and the best strategies of the basic model using method coverage and statement coverage. This indicates that strategies with differentiated p values using method coverage can even be a *cheaper but better* alternative choice for prioritization techniques using statement coverage.

H. Summary and Implications

We summarize the main findings of our experimental study:

- For a wide range of p values (i.e., between 0.95 and 0.50), strategies in both our basic and extended models (on average) outperform or are at least competitive with the *additional* strategy using any combination of test-case and coverage granularities.
- Strategies in the extended model are generally more effective than strategies in the basic model, especially when the values of p are close to 0.
- Strategies in the basic and extended models are more beneficial for method coverage than statement coverage.
- Our extended model is more beneficial for test suites at the test-class level, while our basic model is more suitable for test suites at the test-method level.
- All our strategies using differentiated p values statistically significantly outperform the *additional* strategies using method coverage. Some of our strategies using differentiated p values with method coverage even statistically significantly outperform the *additional* strategies using statement coverage.

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

IV. Related Work

Since there is a considerable amount of research focusing on various issues in test-case prioritization, we partition and discuss the investigated issues into the following categories.

Prioritization Strategies. The *total* and *additional* strategies are the most widely-used prioritization strategies [26]. As neither can always achieve the optimal ordering of test cases [26], researchers have also investigated various other generic strategies. Li et al. [16] present the 2-optimal strategy (a greedy algorithm based on the k-optimal algorithm [17]), a strategy based on hill-climbing, and a strategy based on genetic programming. Jiang et al. [12] present the adaptive random strategy. According to the reported empirical results, the *additional* strategy is more effective than the *total* strategy on average, and other strategies falls between the *total* and *additional* strategies in terms of effectiveness. In this paper, we investigate strategies with flavors of both the *total* and *additional* strategies. Most of our strategies are more effective than either the *total* or the *additional* strategy. 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 [26], function coverage [7], [9], block coverage [6], modified condition/decision coverage [13], method coverage [6] and statically-estimated method coverage [20], [32]. There has also been work on incorporating information on the probability of exposing faults into criteria [9]. There is research [15] on test-case prioritization using coverage of system models, which can be acquired before the coding phase. Mei et al. [22] investigate criteria based on dataflow coverage [21] for testing service-oriented software. In this paper, we investigate generic strategies that can work with any coverage criteria.

Constraints. In practice, there are many constraints affecting test-case prioritization. Elbaum et al. [8] and Park et al. [23] investigate the constraints of test cost and fault severity. Hou et al. [10] investigate the quota constraint on test-case prioritization. Kim and Porter [14] investigate the resource constraint that may not allow the execution of the entire test suite. Walcott et al. [28] and Zhang et al. [33] investigate time constraints that require the selection of a subset of test cases for prioritization. Do et al. [4] 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 strategies may also be applicable.

Usage Scenarios. Prioritized regression test cases can be used for either a specific subsequent version or a number of

subsequent versions. Elbaum et al. [7], [9] 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 [27]) have investigated techniques for only version-specific prioritization. There are also researchers (such as Elbaum et al. [7], [9]) investigating the use of general test-case prioritization techniques in version-specific prioritization. Like other general prioritization techniques, our generic test-case prioritization strategies may also be applicable in version-specific prioritization. Furthermore, it should also be possible to develop a version-specific technique similar to Srivastava and Thiagarajan's technique by using our strategies on coverage of changed code instead of all the code.

V. Conclusion and Future Work

In this paper, we show how the *total* and *additional* strategies can be seen as two extreme instances in models of generic prioritization strategies. Naturally, there is a spectrum of generic strategies between the *total* and *additional* strategies in our models. We also proposed extensions to enable the use of differentiated p values for methods. Empirical results demonstrate that wide ranges of strategies in both our basic and extended models are more effective than either the *total* or the *additional* strategies. Also, wide ranges of our strategies using method coverage can be as effective as or more effective than the *additional* strategy using statement coverage. In a broader sense, we view our results as a fundamental step toward controlling the uncertainty of fault detection in test-case prioritization. In this sense, our models provide a new dimension for creating better prioritization techniques.

In future work, we plan to address the difference between recorded coverage information and actual coverage information in test-case prioritization. We also plan to investigate adaptive test-case prioritization, which changes the probability value of each unit based on actual fault revealing behavior during test case prioritization in regression testing.

Acknowledgements

This research is sponsored in part by the National 973 Program of China No. 2009CB320703, the Science Fund for Creative Research Groups of China No. 61121063, and the National Natural Science Foundation of China under Grant Nos. 91118004, 61228203, and 61272157. This research is also supported by the US NSF through awards CCF-0845628, CNS-0958231, CNS-0720757, and the Air Force Office of Scientific Research through award FA9550-10-1-0406.

References

[1] J. H. Andrews, L. C. Briand, and Y. Labiche, "Is mutation an appropriate tool for testing experiments?" in *ICSE*, 2005, pp. 402–411.

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

[3] H. Do, S. Elbaum, and G. Rothermel, "Supporting controlled experimentation with testing techniques: An infrastructure and its potential impact," *ESE*, vol. 10, no. 4, pp. 405–435, 2005.

[4] H. Do, S. Mirarab, L. Tahvildari, and G. Rothermel, "An empirical study of the effect of time constraints on the cost-benefits of regression testing," in *FSE*, 2008, pp. 71–82.

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

[6] H. Do, G. Rothermel, and A. Kinneer, "Empirical studies of test case prioritization in a JUnit testing environment," in *ISSRE*, 2004, pp. 113–124.

[7] S. Elbaum, A. Malishevsky, and G. Rothermel, "Prioritizing test cases for regression testing," in *ISSTA*, 2000, pp. 102–112.

[8] S. Elbaum, A. Malishevsky, and G. Rothermel, "Incorporating varying test costs and fault severities into test case prioritization," in *ICSE*, 2001, pp. 329–338.

[9] S. Elbaum, A. Malishevsky, and G. Rothermel, "Test case prioritization: A family of empirical studies," *TSE*, vol. 28, no. 2, pp. 159–182, 2002.

[10] S.-S. Hou, L. Zhang, T. Xie, and J. Sun, "Quota-constrained test-case prioritization for regression testing of service-centric systems," in *ICSM*, 2008, pp. 257–266.

[11] D. Jeffrey and N. Gupta, "Improving fault detection capability by selectively retaining test cases during test suite reduction," *TSE*, vol. 32, no. 2, pp. 108–123, 2007.

[12] B. Jiang, Z. Zhang, W. K. Chan, and T. H. Tse, "Adaptive random test case prioritization," in *ASE*, 2009, pp. 257–266.

[13] J. A. Jones and M. J. Harrold, "Test-suite reduction and prioritization for modified condition/decision coverage," in *ICSM*, 2001, pp. 92–101.

[14] J. M. Kim and A. Porter, "A history-based test prioritization technique for regression testing in resource constrained environments," in *ICSE*, 2002, pp. 119–129.

[15] B. Korel, L. Tahat, and M. Harman, "Test prioritization using system models," in *ICSM*, 2005, pp. 559–568.

[16] Z. Li, M. Harman, and R. Hierons, "Search algorithms for regression test case prioritisation," *TSE*, vol. 33, no. 4, pp. 225–237, 2007.

[17] S. Lin, "Computer solutions of the travelling salesman problem," *Bell System Technical Journal*, vol. 44, no. 5, pp. 2245–2269, 1965.

[18] Y.-S. Ma, J. Offutt, and Y. R. Kwon, "MuJava : An automated class mutation system," *STVR*, vol. 15, no. 2, pp. 97–133, 2005.

[19] T. McCabe, "A complexity measure," *TSE*, no. 4, pp. 308–320, 1976.

[20] H. Mei, D. Hao, L. Zhang, L. Zhang, J. Zhou, and G. Rothermel, "A static approach to prioritizing junit test cases," *TSE*, vol. 38, no. 6, pp. 1258–1275, 2012.

[21] L. Mei, W. K. Chan, and T. H. Tse, "Data flow testing of service-oriented workflow applications," in *ICSE*, 2008, pp. 371–380.

[22] L. Mei, Z. Zhang, W. K. Chan, and T. H. Tse, "Test case prioritization for regression testing of service-oriented business applications," in *WWW*, 2009, pp. 901–910.

[23] H. Park, H. Ryu, and J. Baik, "Historical value-based approach for cost-cognizant test case prioritization to improve the effectiveness of regression testing," in *SSIRI*, 2008, pp. 39–46.

[24] X. Qu, M. B. Cohen, and G. Rothermel, "Configuration-aware regression testing: an empirical study of sampling and prioritization," in *ISSTA*, 2008, pp. 75–86.

[25] G. Rothermel, M. J. Harrold, J. von Ronne, and C. Hong, "Empirical studies of test-suite reduction," *STVR*, vol. 12, no. 4, pp. 219–249, 2002.

[26] G. Rothermel, R. H. Untch, C. Chu, and M. J. Harrold, "Test case prioritization: an empirical study," in *ICSM*, 1999, pp. 179–188.

[27] A. Srivastava and J. Thiagarajan, "Effectively prioritizing tests in development environment," in *ISSTA*, 2002, pp. 97–106.

[28] K. R. Walcott, M. L. Soffa, G. M. Kapfhammer, and R. S. Roos, "Time aware test suite prioritization," in *ISSTA*, 2006, pp. 1–11.

[29] L. J. Williams and H. Abdi, "Fisher's least significance difference (LSD) test," in *Encyclopedia of Research Design*. Thousand Oaks, 2010, pp. 491–494.

[30] W. Wong, J. Horgan, S. London, and H. Agrawal, "A study of effective regression testing in practice," in *ISSRE*, 1997, pp. 230–238.

[31] S. Yoo, M. Harman, and S. Ur, "Measuring and improving latency to avoid test suite wear out," in *ICSTW*, 2009, pp. 101–110.

[32] L. Zhang, J. Zhou, D. Hao, L. Zhang, and H. Mei, "Prioritizing JUnit test cases in absence of coverage information," in *ICSM*, 2009, pp. 19–28.

[33] L. Zhang, S.-S. Hou, C. Guo, T. Xie, and H. Mei, "Time-aware test-case prioritization using integer linear programming," in *ISSTA*, 2009, pp. 213–224.