A Uniform Representation of Hybrid Criteria for Regression Testing

Sreedevi Sampath, *Member*, *IEEE*, Renée Bryce, *Member*, *IEEE*, and Atif M. Memon, *Senior Member*, *IEEE*

Abstract—Regression testing tasks of *test case prioritization*, *test suite reduction*/*minimization*, and *regression test selection* are typically centered around *criteria* that are based on code coverage, test execution costs, and code modifications. Researchers have developed and evaluated new individual criteria; others have combined existing criteria in different ways to form what we—and some others—call *hybrid* criteria. In this paper, we formalize the notion of combining multiple criteria into a hybrid. Our goal is to create a uniform representation of such combinations so that they can be described unambiguously and shared among researchers. We envision that such sharing will allow researchers to implement, study, extend, and evaluate the hybrids using a common set of techniques and tools. We precisely formulate three hybrid combinations, *Rank*, *Merge*, and *Choice*, and demonstrate their usefulness in two ways. First, we recast, in terms of our formulations, others' previously reported work on hybrid criteria. Second, we use our previous results on test case prioritization to create and evaluate new hybrid criteria. Our findings suggest that hybrid criteria of others can be described using our *Merge* and *Rank* formulations, and that the hybrid criteria we developed most often outperformed their constituent individual criteria.

Index Terms—Test case prioritization, test criteria, hybrid test criteria, web testing, GUI testing

1 Introduction

Regression testing activities of *test case prioritization* [3], [63], *test suite reduction* (also called *minimization*) [21], [25], [28], [36], [68], and *regression test selection* [41] are typically centered around *criteria* that determine "which test cases to select" or "which test case to execute next." In test case prioritization, test cases are ordered based on a certain criterion and test cases with highest priority are executed first to achieve a performance goal. In test suite reduction/minimization (in the rest of the paper, for simplicity, we will use the term "reduction"), test cases that become redundant over time are removed from the test suite to create a smaller set of test cases. In regression test selection, a subset of test cases is selected from a larger original suite.

For example, Rothermel et al. [50] select test cases that yield the greatest statement coverage; Sprenkle et al. [58] select test cases that cover each executed statement at least once; in both of these examples, the selection criteria are embodied in code coverage criteria-covering program statements. Other criteria consider program modifications [59] and cost [56]. There are numerous other examples of criteria used for prioritization [3], [9], [12], [14], [15], [16], [29], [33], [63], [71], reduction [2], [21], [22], [25], [28], [31], [39], [48], [53], [65], [68], and selection [41] (Yoo and Harman [69] provide an extensive survey of regression techniques).

Several researchers combine multiple criteria and show that the combination is more useful than the individual criteria. For instance, Jeffrey et al. [30] use *branch* as a primary criterion and *all-uses* as a secondary criterion for test suite reduction—if a test is redundant with respect to the primary criterion, it may still be included in the reduced test suite if it is not redundant with respect to the secondary criterion. Similarly, Lin et al. [36] combine a primary criterion of *branch* with a secondary criterion of *def-use* for test suite reduction. They employ the secondary criterion only when the use of the primary criterion returns multiple test cases that satisfy its coverage requirement. On the other hand, Black et al. [2] *simultaneously* use two criteria ("*all-uses*" and "*error detection rates from previous runs*") for test suite reduction by combining them using binary integer linear programming (ILP). Other researchers[1] also combine criteria in different ways and report that such combinations are useful. However, researchers use different representations, if any, to formulate and describe the multiple criteria. Also, very few have empirically compared their new combination with prior published ones. Although there are many reasons why researchers do not perform such comparisons (e.g., lack of tools and shared artifacts), we feel that one big reason for not doing so is the lack of a standard representation to formulate, describe, and understand the new multicriteria.

- *S. Sampath is with the Department of Information Systems, University of Maryland Baltimore County, 1000 Hilltop Circle, Baltimore, MD 21250. E-mail: sampath@umbc.edu.*
- *R. Bryce is with the Department of Computer Science and Engineering, University of North Texas, 1155 Union Circle #311366, Denton, TX 76203-5017. E-mail: Renee.Bryce@unt.edu.*
- *A.M. Memon is with the Department of Computer Science, University of Maryland College Park, A.V. Williams Building, College Park, MD 20742. E-mail: atif@cs.umd.edu.*

Manuscript received 16 Sept. 2012; revised 14 Jan. 2013; accepted 3 Mar. 2013; published online 20 Mar. 2013.
Recommended for acceptance by G. Rothermel.
For information on obtaining reprints of this article, please send e-mail to: tse@computer.org, and reference IEEECS Log Number TSE-2012-09-0262.
Digital Object Identifier no. 10.1109/TSE.2013.16.

1. Of the 593 research papers that we examined for this research, a total of 44 papers have combined criteria in different ways [1], [2], [3], [4], [7], [8], [10], [11], [14], [17], [18], [19], [21], [23], [24], [25], [26], [27], [28], [30], [32], [34], [35], [36], [37], [38], [40], [41], [42], [44], [45], [47], [52], [54], [60], [61], [63], [64], [66], [67], [68], [70], [72], [73].

In this paper, we formalize the notion of combining multiple criteria together into what we call a *hybrid*. We create a uniform representation of such combinations so that they may be described unambiguously. We envision that researchers who use our representation will be able to share these descriptions with others, perhaps in a shared repository, and thereby facilitate replication and comparative studies between hybrids. We see increasing interest in using hybrid criteria for regression testing (71 percent of the 44 papers were published in the last 5 years), so we feel that this research is both timely and relevant.

Because hybrid criteria may be obtained by combining individual criteria in various ways, our representation needs to be very general. To allow generality, we use the functional paradigm—individual criteria are represented as operands and their combination is represented as an operator. We use function composition, and its well-known semantics, to compose different criteria together, giving us great flexibility in how we can combine and use the criteria, including "hybrids of hybrids" via nested compositions of functions.

We describe three ways of combining criteria: 1) *Rank* 2) *Merge*, and 3) *Choice*. In *Rank*, the criteria being combined are ranked *in order of importance*; the first criterion is applied first; a user-supplied function determines if the second and all subsequent criteria, in order, need to be used. Depending on the situation, *Rank* may use several criteria but they must be applied one by one, in order of importance. A more important criterion must be applied before one of less importance is applied. *Merge*, on the other hand, allows for multiple criteria to be considered *simultaneously*; all criteria are *combined* first and the combination is used. Finally, *Choice* selects *only one* from a set of equally important criteria using a user-supplied selection function. We acknowledge that these three ways of combining criteria are by no means complete. They are meant to serve as a starting point for more work in the field; indeed, we expect to create additional ways to combine criteria; this is a subject for future research.

We demonstrate the usefulness of our three formulations in two ways. First, we recast others' previously reported work on combining multiple criteria for regression testing in terms of our hybrid formulation. Second, we use our past empirical results on test case prioritization [3] to demonstrate the creation and evaluation of new hybrid criteria. In previous work, we used individual criteria. We demonstrate how one would combine the individual criteria and evaluate new hybrid criteria, explaining the strengths and weaknesses of the combinations.

Our paper makes the following contributions:

1. We formalize the notion of combining multiple criteria for regression testing, giving researchers a new way to represent such combinations.
2. Although we provide three examples of operators, our overall function-based representation is general, allowing future researchers to create new hybrid criteria.
3. We clearly place others' past work in terms of *Rank* and *Merge* operators.
4. We show examples of how one could combine criteria in the domain of web and graphical-user interface (GUI)-based applications.
5. We empirically evaluate the benefits of combining criteria for these domains.

Our findings suggest that others' prior reported work can be described in terms of only two of our operators: *Rank* and *Merge*. This gives us a fair amount of confidence that these are important operators for hybrid formulations. We also observed that the hybrid criteria we developed most often outperformed their constituent individual criteria.

The remainder of this paper is organized as follows: Section 2 summarizes existing work on combining multiple criteria. Section 3 presents our three formulations of hybrid. Section 4 demonstrates the use of our formulations. Finally, Section 5 concludes with a discussion of future work.

2 Background and Related Work

The use of hybrid criteria (also called "multicriteria" [41], "multi-objective techniques" [68], and "breaking ties" [36]) has been motivated by multiple researchers over the years. Most of these researchers agree that locating faults is a complex process; the use of a single criterion severely limits the ability of the resulting regression test suite to locate faults [41]. Multiple contextual factors that are impossible to account for via a single criterion, for instance, the characteristics of an application, modifications made since the last testing cycle, and original test suite, may influence whether a particular criterion is well suited for a particular regression testing situation. Multiple criteria have the potential to improve the effectiveness of regression testing techniques over those with a single criterion. A large number of studies have successfully shown this to be true [1], [2], [3], [4], [7], [8], [10], [11], [14], [17], [18], [19], [21], [23], [24], [25], [26], [27], [28], [30], [32], [34], [35], [36], [37], [38], [40], [41], [42], [44], [45], [47], [52], [54], [60], [61], [63], [64], [66], [67], [68], [70], [72], [73]. Further, Harman [20] motivates the use of multiple criteria for both our paper and future work. He generalizes the regression testing problem into a multiobjective problem and provides a list of criteria that are useful in industry. While different criteria may have different weights, he recommends using Pareto optimization when weights will not work. The paper provides many opportunities for future work such as experimentation.

In this section, we review a representative sample of these existing techniques that use multiple criteria for various regression testing tasks, i.e., reduction [2], [21], [25], [30], [36], [68], prioritization [3], [63], and selection [41].

As early as Harrold et al. [21] recognized, in what subsequently became known as the "HGS[2] paper," that multiple criteria may be used simultaneously for test suite reduction. Although the "HGS algorithm" has mostly been used with a single criterion, their original paper explicitly mentions how it may be used with multiple criteria. They outline three situations for using two criteria: 1) test cases that satisfy the first criterion form a proper subset of test cases that satisfy the second criterion, 2) test cases that satisfy the first criterion overlap with test cases that satisfy

2. Abbreviated from the researchers' last names: *H*arrold, *G*upta, *S*offa.

the second criterion, and 3) there is no overlap between test cases that satisfy the two criteria. They mention how their algorithm may be used in these situations. However, they do not evaluate their multicriteria approach.

Jeffrey and Gupta [30] develop an approach that uses multiple criteria for test suite reduction. In their approach, if a test is deemed redundant by the first criterion, they check whether the test is also redundant according to the second criterion. If not, then the test is included in the reduced suite. They conduct experiments using the seven *Siemens programs* and the *Space program*. In their experiments, they use the criteria branch coverage, all-uses, and subpaths of length 3. They measure reduction in test suite size, loss in fault detection effectiveness, and additional-faults-to-additional-tests ratio. They also conduct another experiment using four Java programs (*bst*, *avl*, *heap*, *sort*), where each program contains a single method operating on a data structure. In this experiment, they use black-box criteria as primary and branches and def-use coverage as secondary criteria. They report that using multiple criteria helps identify tests that expose different faults, and that multiple criteria increase the fault detection effectiveness of the reduced suite.

Sampath et al. [52] combine multiple criteria for test suite reduction. In their work, they propose two hybrids. In one case, they merge the program coverage and usage-based requirements and supply the combined matrix to the HGS algorithm. In the second case, they use the HGS algorithm with program-coverage-based criteria as primary, and usage-based criteria as *tie breakers*, i.e., to select one of several test cases that the primary criterion may consider *equivalent*. They use statement, method, and conditional coverage as the program-coverage-based criteria, and "base requests," "base requests and name," "base requests and name-value," "sequences of base requests of size 2," and "sequences of base requests and name of size 2" as usage-based criteria. They evaluate their approach using one web application with seeded faults. Their findings show that using usage-based criteria as tie breakers produces more effective test suites with respect to program coverage and fault detection than without using them.

Lin and Huang [36] perform regression test suite reduction with their *reduction with tie-breaking (RTB)* approach that extends the HGS algorithm [21] and the "GRE algorithm" [5], [6] by using two criteria. When ties are encountered between test cases as per the primary criterion, their technique uses a secondary criterion to break ties. In the evaluation of their approach, they use the seven *Siemens programs* with seeded faults and the *Space program* with natural faults. Their primary criterion is branch coverage, and their secondary criterion is def-use pair coverage. They measure reduction in test suite size, loss of fault detection effectiveness, and faults-to-test ratio in their experiments. They find that by integrating RTB with HGS [21] and GRE [6], the fault detection effectiveness can be improved slightly without significantly affecting the size of the reduced suites.

Walcott et al. [63] present an approach for time-aware test case prioritization using two criteria: 1) execution time and 2) code coverage (in particular, block and method coverage). Their approach uses a genetic algorithm to prioritize the regression test suites based on these two criteria. In their experiments, they use two applications, *Gradebook* and *JDepend* with mutation faults. They find that their time-aware prioritizations outperform other prioritization techniques.

Yoo and Harman [68] perform test suite reduction using a genetic algorithm that uses two- and three-objective formulations. They use code coverage and execution time in the two-objective formulation, and add fault detection as the third objective in the three-objective formulation. Their goal is to select a *Pareto efficient* subset of the test suite based on satisfying the multiple criteria. They compare their genetic algorithm to a baseline *additional-greedy* algorithm. They supply the results from the additional greedy algorithm as the initial population to the genetic algorithm. They empirically examine five subject applications seeded with faults, *flex*, *grep*, *gzip*, *sed*, and *space*, obtained from SIR [55]. For three of these applications, they observe that the multi-objective genetic algorithm finds solutions not reported by the additional-greedy algorithm, whereas for the other systems, the genetic algorithm could not improve upon the solutions of additional greedy.

Black et al. [2] perform test suite reduction using a combination of two criteria: 1) all-uses and 2) error detection rates from previous runs of regression testing. They develop a binary integer linear programming model based on the two criteria. Weights on the criteria allow one criterion to take precedence over the other. Their experiments on seven programs from the *Siemens* suite that are seeded with faults find that this approach successfully selects test cases that reveal faults in subsequent versions of the program.

Hsu et al. [25] perform test suite reduction using multiple criteria modeled as an integer linear programming problem. They propose three policies by which the criteria can be combined: 1) weighted, 2) prioritized, and 3) hybrid. In the weighted policy, they give a weight to each criterion/objective and consider all the criteria together during reduction. In the prioritized policy, they assign priorities to criteria and select them one at a time for prioritization. The third policy, hybrid, divides objectives into groups with weights on the individual objectives, and assigns priorities to the groups. In their experiments, they evaluated their approach using the seven programs from the *Siemens* suite and three additional programs, *flex*, *LogicBlox*, and *Eclipse*, also from SIR [55]. They used a combination of seeded and real faults in these systems. They report that their approach converged to a solution quickly for most reduction problems, and it performed well or better than the HGS [21] algorithm in terms of size of the minimized test suite and time to generate the test suite.

Mirarab et al. [41] perform test case selection using two coverage-based criteria. The criteria they use are 1) maximize total coverage of software elements of all test cases, and 2) maximize the minimum coverage across all software elements. These two criteria are used to formulate the test selection problem as an integer linear programming problem. They also propose a greedy algorithm that seeks to maximize the minimum sum of coverage across all

TABLE 1
Five Example Test Cases, Their Coverage and Other Metrics, and Faults Detected

Test Cases	Events Covered			Statements Covered					Branches		Exec.	Length	Faults Detected				
ID	e_1	e_2	e_3	s_1	s_2	s_3	s_4	s_5	b_1	b_2	*(sec.)*	*(# events)*	f_1	f_2	f_3	f_4	f_5
T_1	0	1	1	1	0	1	1	1	0	0	0.5	5	0	0	1	0	1
T_2	1	0	1	0	1	0	1	0	0	0	0.1	3	0	1	0	1	0
T_3	1	1	0	1	0	1	0	1	1	1	0.8	2	1	0	1	0	0
T_4	0	0	1	0	1	0	1	0	1	0	0.1	5	0	1	0	0	0
T_5	1	1	0	1	0	0	0	0	0	1	0.9	6	0	0	0	0	0
	$\mathcal{E}$			$\mathcal{S}$					$\mathcal{B}$		$\mathcal{X}$	$\mathcal{L}$	$\mathcal{F}$				

software elements. They use five open source Java programs from SIR [55] that contain mutation faults. They find that their proposed technique is quite effective at detecting faults when compared to other approaches, such as a Bayesian network-based, time-aware test case prioritization [63], and method coverage-based techniques.

In our previous work [3], we present a hybrid approach for test case prioritization using two criteria: 1) frequency-based criteria and 2) a combinatorial criterion, two way. We prioritize test cases by one criterion and change to a second criterion when the *average percent of faults detected* (APFD) does not increase after a specified number of test cases. In our evaluation, we use one web application with seeded faults. We found that in several cases, the hybrid prioritized test orders outperformed the test orders prioritized by individual criteria.

A large fraction of the 44 "hybrid papers" that we examined in detail for this research mention the benefits of using multiple criteria over a single criterion for regression testing. However, each paper uses its own notation, if any, when formulating the multiple criteria regression testing problem. Also, very few have empirically compared their new approach with any prior approaches. Although there are many reasons why researchers do not perform such comparisons, we feel that one big reason for not doing so is the lack of standard representations available to researchers to formulate and describe their new multicriteria approaches. In this paper, we provide a starting point for doing so. We describe a new way to represent multicriteria approaches. To exemplify its use, we recast previous approaches in terms of this representation. Moreover, we walk the reader through a detailed example of how we use the representation to extend our own prior work on test case prioritization of event-driven systems.

3 Defining Hybrid Criteria

In this section, we formally define the three ways in which we develop hybrid criteria for performing regression testing tasks. Because our hybrid criteria are obtained by combining individual criteria in various ways, we formulate the hybrid using the functional paradigm. This allows us to use function composition, and its well-known semantics, to compose different criteria together, giving us great flexibility in how we can combine and use the criteria.

We use a running example, seen in Table 1, of a test suite to demonstrate the use of our function formulation as well as the hybrid combinations. The suite contains five test cases, each of which is a sequence of events; the table shows the various measured characteristics of these test cases, including the events, statements, and branches they cover, their execution time, length, and faults they detect. Without loss of generality, we assume that these are represented as matrices (e.g., coverage matrix $\mathcal{S}$ for statements covered) and as vectors of values (e.g., $\mathcal{L}$ for length of the test cases). These two representations of test cases are commonly used by software testing tools [62] as well as researchers [74].

To provide focus, we will restrict our primary thread of discussion to the problem of test case prioritization. At certain points in the discussion, however, we will discuss extensions to test suite reduction and regression test selection. We feel that this approach helps to simplify the flow of this paper.

3.1 Representing Stand-Alone Criteria

We start by defining existing conventional, stand-alone (nonhybrid) criteria as functions. Because of our focus on test case prioritization, one of the key components of our formulation is a function *next()* that takes three parameters: 1) the sequence of test cases selected thus far, 2) the complete test suite, 3) a matrix (or vector) encoding the relationship between all test cases and the metric being used to compute the prioritized order (e.g., statement coverage) paired with a function *g()* to be used for the computation. The output of *next()* is a set of test cases that are next in the prioritized order.

A typical usage of *next()* would start with an empty sequence as the first parameter; *next()* would return the highest priority test cases; these would be used in the second invocation as the first parameter to obtain the next most important test cases; subsequent iterative invocations would pass—as the first parameter—all ordered test cases obtained thus far to obtain the next important test cases; the iterations will continue until all tests have been ordered.

Consider our running example of Table 1. Suppose we want to prioritize our five test cases using the classical *additional statement coverage prioritization* [50] technique, which *"iteratively selects a test case that yields the greatest statement coverage, then adjusts the coverage information on all remaining test cases to indicate their coverage of statements not yet covered, and repeats this process until all statements covered by at least one test case have been covered."* When multiple test cases cover the same number of statements not yet covered, a random choice of one test case is made. We define function *g()* for additional coverage prioritization shown in Fig. 1.

Let us use this $g()$ to prioritize the test cases in our running example. Our first invocation is $next([], \{T_1, T_2, T_3, T_4, T_5\}, (\mathcal{S}, g()))$, which returns T_1, the test case that covers the maximum number, in this case 4, of statements.

```
Require: Seq //the test cases ordered so far
Require: Suite //the complete test suite
Require: Cov //the coverage matrix or vector
 1: x ← coverage elements covered by tests in Seq
 2: if isMatrix(Cov) then
 3:    t ← delete columns for x from Cov
 4:    if hasNoColumns(t) then
 5:       return φ
 6:    end if
 7: else
 8:    t ← Cov
 9: end if
10: y ← delete rows for tests in Seq from t
11: updateScores()
12: if hasNoRows(y) then
13:    return φ
14: end if
15: Set ← findall tests that cover largest elements in y
16: return Set
```

Fig. 1. $g()$ for additional coverage prioritization.

```
Require: Seq //the test cases ordered so far
Require: Suite //the complete test suite
Require: e //the criterion and g() pair
 1: y ← Seq
 2: done ← FALSE
 3: while not(done) do
 4:    x ← next(y, Suite, e)
 5:    if empty(x) then
 6:       done ← TRUE
 7:    else if singleElement(x) then
 8:       y ← append x to y
 9:    else
10:       y ← append RandomElt(x) to y
11:    end if
12: end while
13: y ← append unused elements of Suite to y
14: return y
```

Fig. 2. Invocation function.

Our next invocation is $next([T_1], \{T_1, T_2, T_3, T_4, T_5\}, (\mathcal{S}, g()))$, which returns $\{T_2, T_4\}$, the two test cases that cover the remaining statement s_2. We randomly pick test case T_4, and invoke $next([T_1, T_4], \{T_1, T_2, T_3, T_4, T_5\}, (\mathcal{S}, g()))$, which returns an empty set, indicating that all statements have been covered. We can then add the remaining test cases $\{T_2, T_3, T_5\}$ in any order.

We note that a simple function is needed to manage all of the above invocations of *next()*, make the random choices, and add remaining test cases. The pseudocode for this *invocation* function is shown in Fig. 2. The function starts with Seq=[], invokes $next()$ (Line 4); if it gets multiple test cases, it makes a random choice (Line 10); it iterates until no more tests are returned (Line 6). It finally appends the unused test cases to the returned prioritized sequence (Line 13).

Our functional formulation allows us to quickly realize alternative test case prioritization techniques by developing new invocations and $g()$ functions that we can "plug into" the overall prioritization framework. For example, in Line 13, if we order the remaining test cases by reapplying additional coverage prioritization, i.e., by resetting the coverage vectors for all of these test cases to their initial values, and reapplying the algorithm ignoring all previously prioritized test cases, we would have an implementation of the prioritization technique developed by Rothermel et al. [50].

In this example, we restrict the $g()$ function to operate on a single coverage matrix/vector. This allows us to reuse the same $g()$ function in many places. Our invocation function provides high-level control over the use of the $next()$ function. This separation of roles between the invocation and $g()$ functions is deliberate; we will continue to use this separation in the remainder of the paper.

It is easy to see how the same formulation may be modified for regression test selection and test suite reduction. If we delete Line 13 in the invocation function, Fig. 2, the remaining unused test cases are discarded from the selected suite in y. In this case, all the coverage elements are covered by the test cases already selected. Hence, the final suite, stored in y, is the *reduced* form of Suite because it covers exactly the same coverage elements. Test suite minimization, on the other hand, requires a slightly more involved change. It requires that we define a new, nongreedy $g()$ function so that a *minimized* suite is obtained. However, our fundamental infrastructure remains the same.

3.2 Representing Second-Order Criteria

We now extend our formulation to consider multiple criteria, developing what we term as *second-order* criteria. As can be imagined, criteria may be combined in a multitude of ways to form a hybrid. At this point, we step back and consider the philosophy behind creating hybrids for test case prioritization. The key idea behind the hybrid is that it combines multiple criteria, yielding, at every decision—*which test case to select next*—of the prioritization algorithm, the strengths of the constituent individual criteria. One possible hybrid is to order the individual criteria (e.g., as primary, secondary, tertiary, and so on), start with the primary, and use the secondary only if the primary produces ties in test case selection, i.e., multiple test cases satisfy the primary criterion. This suggests a ranking type of hybrid. Another possibility is to *mathematically combine* the individual criteria into a single criterion. For instance, let us say that we want to order tests by their ability to cover the sum of statement and branch coverage, i.e., the most important test case covers the most statements and branches combined. This suggests that we need a hybrid that can combine criteria together and apply the combination simultaneously. Finally, a third type of hybrid should allow a clear selection between multiple criteria based on a selection function.

Building upon our above intuition, in this paper, we develop three ways, 1) *Rank*, 2) *Merge*, and 3) *Choice*, to combine criteria. As mentioned earlier, we recognize that this set is not complete. Developing additional ways is a subject for future research.

Rank: Intuitively, the criteria being combined are *ranked* in order of importance and applied in series. For example, one implementation of *Rank* could use the first criterion to prioritize the test cases; only when that criterion fails (as determined by function $g()$) or produces ties is the second criterion used; if that too fails, the third, and all subsequent criteria in order, are used.

Consider our previous example of additional statement coverage prioritization. Let us modify the prioritization approach so that statement coverage is used as a primary criterion, but event coverage, encoded in $\mathcal{E}$, is used as a secondary criterion. Our first invocation is modified to $next([], \{T_1, T_2, T_3, T_4, T_5\}, Rank((\mathcal{S}, g()), (\mathcal{E}, g())))$. We execute this invocation as follows: We first invoke $next([], \{T_1, T_2, T_3, T_4, T_5\}, (\mathcal{S}, g()))$, which is the base case presented earlier. It returns T_1. As before, we invoke $next([T_1], \{T_1, T_2, T_3, T_4, T_5\}, (\mathcal{S}, g()))$, which returns $\{T_2, T_4\}$. Because we have a tie, we use our second criterion, event coverage in an invocation $next([T_1], \{T_2, T_4\}, (selectRows(\{T_1, T_2, T_4\}, \mathcal{E}), g()))$, where $selectRows(subsetOfRows, Matrix)$ gives the submatrix, consisting only of the specified subset of rows from the matrix. This invocation returns T_2 because it covers event e_1 not covered by T_1. We then invoke $next([T_1, T_2], \{T_1, T_2, T_3, T_4, T_5\}, (\mathcal{S}, g()))$, which returns an empty set. This causes us to use our secondary criterion again and we invoke $next([T_1, T_2], \{T_1, T_2, T_3, T_4, T_5\}, (\mathcal{E}, g()))$, which also returns an empty set because all events have been covered by T_1 and T_2. Because we have no more criteria, we are free to select the remaining test cases $\{T_3, T_4, T_5\}$ in any order.

Let us suppose that we had specified test case length as a *tertiary* criterion, invoking $next([], \{T_1, T_2, T_3, T_4, T_5\}, Rank((\mathcal{S}, g()), (\mathcal{E}, g()), (\mathcal{L}, g())))$. Ties between $\{T_3, T_4, T_5\}$ would be resolved by the tertiary criterion, picking the longest test case first, namely T_5, followed by T_4, and T_3.

In this example, we showed several important things. First, a user specifies the rank order of the criteria using a $Rank()$ function. Second, each criterion is associated with its own computation function $g()$. In our case, we were able to use the same function for all criteria but this may not always be the case. Third, the evaluation of the invocation of $next()$ involving $Rank()$ is performed in a top-down manner, where each individual criterion with its associated function is invoked individually in order. Fourth, we seamlessly mixed the use of our matrix and vector representations because of the way we defined $g()$. This may not always be possible—at the very least, one may need to provide ways to convert between *types* of representations, for example, vector and matrix. Finally, the less important criteria are used by the invocation function only when the more important ones have failed to provide an adequate solution. In our example, ties and empty set returns were considered to be inadequate solutions. Similar tie-breaking approaches have been proposed in the past [36], [52].

As before, the overall computation is controlled by an invocation function. The pseudocode for the invocation function that we used in the above example is shown in Fig. 3. The code selects the first element from the `Rank` specification (Line 5), and invokes *next()* using that element (Line 8). If a single test case is returned (Line 9), then it is

```
Require: Seq //the test cases ordered so far
Require: Suite //the complete test suite
Require: Rank //the Rank() specification
 1: y ← Seq
 2: done ← FALSE
 3: while not(done) do
 4:    s ← Suite
 5:    e ← first element in Rank
 6:    resolved ← FALSE
 7:    while (not(resolved) && not(done)) do
 8:       x ← next(y, s, e)
 9:       if singleElement(x) then
10:          y ← append x to y
11:          resolved ← TRUE
12:       else
13:          e ← next element in Rank
14:          if null(e) then
15:             done ← TRUE
16:          else
17:             s ← x
18:             Cov(e) ← selectRows(y ∪ x, Cov(e))
19:          end if
20:       end if
21:    end while
22: end while
23: y ← append unused elements of Suite to y
24: return y
```

Fig. 3. The $Rank_{H(c_x, c_y, \ldots, c_z)}$ invocation function.

appended to the test sequences ordered so far (Line 10). The while loop of Line 3 continues in this way as long as a single element is always returned. If multiple (or no) elements are returned by the call at Line 8, then the next element in the `Rank` specification is tried (Line 13). If there are no more elements in `Rank`, then the remaining test cases are added to y in any order (Line 23). If there are additional elements in `Rank` (lines 17-18), then the tie between the elements in x is resolved by temporarily reducing the suite to only the elements involved in the tie (Line 17) and preparing the coverage matrix (vector) part of e, represented as $Cov(e)$ (Line 18). Once the tie is resolved, i.e., when the invocation of *next()* at Line 8 (the algorithm will go back to Line 8 next) returns a single element (Line 9) causing *resolved* to become *TRUE* or there are no more criteria in the `Rank` specification (Line 14), causing *done* to become *TRUE*, the inner while loop terminates. The algorithm loops back to the while loop at Line 3, which reinitializes s to the full suite at Line 4. The outer loop will terminate when all elements in `Rank` are exhausted. Any unused elements from `Suite` can be added in any order. The particular way we used the *Rank* hybrid approach in these two examples, with our specific implementation of $g()$ for criteria $c_x, c_y, \ldots, c_z$, will be termed $Rank_{H(c_x, c_y, \ldots, c_z)}$ in the remainder of this paper.

One can come up with multiple formulations of *Rank* by varying the criteria, the *g()* functions, and the invocation function. Consider, for example, an alternative invocation function that prioritizes test cases by starting with the first specified criterion; it computes the APFD increase of the last

10 percent of test cases already prioritized. If there is a zero increase, it switches to the next specified criterion until all tests have been ordered or no criteria are left, in which case the remaining tests may be added in any order (this approach was in fact used in earlier work [3]).

More formally, given a test suite T, an ordered sequence of criteria $\langle c_1, c_2, c_3, \ldots, c_m \rangle$, and a Boolean function $\mathcal{F}()$, our formulation of *Rank* may be represented as a function $\mathcal{R}(T, \langle c_1, c_2, c_3, \ldots, c_m \rangle, \mathcal{F}())$ that returns a set/sequence of test cases $\{t_1, t_2, t_3, \ldots, t_n\}$ satisfying the following properties:

1. $t_i \in T$, for all $(1 \leq i \leq n)$.
2. $t_i \preceq t_j$, for all $(1 \leq i \leq n)$ and $(i \leq j \leq n)$. The $\preceq$ operator means that when t_i was ordered/selected by criterion c_x and t_j was ordered/selected by criterion c_y, it must necessarily be the case that $x \leq y$, for $(1 \leq x \leq m)$ and $(1 \leq y \leq m)$. This condition is important for *Rank* because earlier tests in the test order must use the criteria that have been specified as more important by the user.
3. For all $(1 \leq i \leq n)$, whenever t_i was ordered/selected by criterion c_x, $(1 \leq x \leq m)$, then $\mathcal{F}(t_i, c_x)$ returns *TRUE* and for all $1 \leq l < x$, $\mathcal{F}(t_i, c_l)$ returns *FALSE*. This condition states that later criteria in the specified order may be used only in the situation that earlier ones have "failed," as determined by the user-supplied function $\mathcal{F}()$.

Merge: Intuitively, this way to combine multiple criteria considers all the criteria *simultaneously* to order the tests. For example, one implementation of *Merge* could use an operator to combine all the matrices (vectors) into one *compound* matrix (vector) and uses that single compound matrix (vector) to order the test cases. This could be achieved, for example, by the operator *ArrayFlatten* (we will abbreviate this to *AF*) in *Mathematica*[3] which, among other operations, joins two or more matrices *horizontally* to make a new matrix. Obviously, the number of rows in all matrices being concatenated must be the same for a sensible resulting matrix.

If we apply *AF* to $\mathcal{S}$ and $\mathcal{B}$ in our running example, we get a matrix with seven columns. We can then use this matrix in an invocation $next([], \{T_1, T_2, T_3, T_4, T_5\}, Merge((\mathcal{S}, \mathcal{B}), g()\,; AF))$. In effect, the most important test case in the prioritized order would be one that covers the maximum *sum* of the number of statements *and* branches. This turns out to be T_3. The next invocation $next([T_3], \{T_1, T_2, T_3, T_4, T_5\}, Merge((\mathcal{S},\ \mathcal{B}), g(); AF))$ yields $\{T_2, T_4\}$ because T_2 and T_4 both cover s_2 and s_4, statements not covered by T_3. A random choice between these two test cases, say T_4, covers all statements and branches; the remaining tests can be added in any order.

Consider the *Merge* specification in the first invocation, i.e., $Merge((\mathcal{S}, \mathcal{B}), g(); AF))$. The invocation function splits $g(); AF$ and applies the function AF to the two matrices $\mathcal{S}$ and $\mathcal{B}$, as $AF(\{\mathcal{S}, \mathcal{B}\})$. The result is then used as a single matrix by $g()$.

This handling of the *Merge* specification, especially the function processing that combines all matrices to yield a single matrix gives us great flexibility in defining different types of hybrid criteria. For example, consider the invocation $next([], \{T_1, T_2, T_3, T_4, T_5\}, Merge((\mathcal{S}, \mathcal{B}), g(); AF \circ (\#1 \times 2, \#2 \times 3)\&))$. Here, the merging is done by the *composed* function $AF \circ (\#1 \times 2, \#2 \times 3)\&$. It is executed as follows: The second function in the composition, i.e., $(\#1 \times 2, \#2 \times 3)\&$ is a *pure function*[4] that explicitly names its two parameters as $\#1$ and $\#2$; these are substituted by the matrices $\mathcal{S}$ and $\mathcal{B}$, essentially resulting in the invocation $(\mathcal{S} \times 2, \mathcal{B} \times 3)$. This invocation multiplies each element in $\mathcal{S}$ by 2, and each element in $\mathcal{B}$ by 3, and returns a pair of matrices. Using the semantics of function composition, these two matrices are then passed as parameters to *AF*, which combines them to produce a single matrix. This type of *Merge* is useful when *weighting* criteria differently. In our example above, branch coverage gets a higher weight, i.e., 3 than statement coverage, which gets a weight of 2.

Our particular implementation of *Merge* together with *AF* and our specific implementation of $g()$ for criteria $c_x, c_y, \ldots, c_z$ will be represented as $Merge_{U(c_x, c_y, \ldots, c_z)}$ in the remainder of this paper. We will use the somewhat cumbersome notation $Merge_{U(c_x:x, c_y:y, \ldots, c_z:z)}$ if we associate numeric weights $(x, y, \ldots, z)$ with the criteria $(c_x, c_y, \ldots, c_z)$.

More formally, given a test suite T, a set of criteria $\{c_1, c_2, c_3, \ldots, c_m\}$, and a merging function $\mathcal{G}()$, our formulation of *Merge* may be represented as a function $\mathcal{M}(T, \{c_1, c_2, c_3, \ldots, c_m\}, \mathcal{G}())$ that returns a set/sequence of test cases $\{t_1, t_2, t_3, \ldots, t_n\}$ satisfying the following properties:

1. $t_i \in T$, for all $(1 \leq i \leq n)$.
2. $t_i \ll t_j$, for all $(1 \leq i \leq n)$ and $(i \leq j \leq n)$. The $\ll$ operator means that $\mathcal{G}(t_i, \{c_1, c_2, c_3, \ldots, c_m\}) \leq \mathcal{G}(t_j, \{c_1, c_2, c_3, \ldots, c_m\})$, i.e., t_i *"is deemed to be better than or same as"* t_j as determined by $\mathcal{G}()$. It is important to note that the merging function is able to simultaneously consider all criteria.

Note that our definition of merge does not force the merge operation to take a set of criteria/objectives and produce a single hybrid—some implementations of merge may do this; however, this is not a necessity. In case the set of criteria/objectives cannot be managed into a single criterion (as in pareto optimality [68]), they may be left as separate. All that merge requires is the ability to compute a function that considers all the criteria.

Choice: Intuitively, our third and final way to combine criteria selects one of a set of criteria using a selection function. Note that unlike *Rank*, each criterion has the opportunity for selection based on coverage criteria; moreover, a previously selected criterion may be reselected by the selection function. The invocation function that implements *Choice* uses all matrices (vectors) that are specified in *Choice* in parallel in separate $next()$ invocations; the individual outcomes of all these $next()$s are recorded. An additional step computes the "goodness" of each result. The choice with the highest goodness is selected as the output. The process is repeated for the remaining unordered tests until all tests have been ordered.

Consider, for example, an invocation function that prioritizes test cases by selecting the 10 percent that yield

3. http://reference.wolfram.com/mathematica/ref/ArrayFlatten.html.
4. Mathematica-like usage.

the best increase in APFD, i.e., the goodness is the APFD increase. The function continues to select test cases in 10 percent blocks until all tests have been ordered. In the remainder of this paper, we will represent this prioritization technique by $Choice_{T(c_x,c_y,\ldots,c_z)}$ for criteria $c_x, c_y, \ldots, c_z$.

Consider another instance of *Choice*—let us suppose, in our running example, that we want to order test cases so that those that cover the maximum percentage of criteria elements, for any of the three criteria: *event*, *statement*, and *branch*, are run first. That is, the goodness is measured by the percentage of elements covered by each test case for each criterion. Test case t_3 covers 100 percent branches; no other test case covers 100 percent of any of the coverage elements. Hence, t_3 is run first. Next, t_1 covers 80 percent of the statements; it is run next. Tests t_2 and t_5 cover 66.66 percent of events; they come next. Finally, t_4 comes last. Our series of invocations are $next([], \{t_1, t_2, t_3, t_4, t_5\}, (Choice(\mathcal{E}, \mathcal{S}, \mathcal{B}), g()))$ that returns t_3, followed by $next([t_3], \{t_1, t_2, t_3, t_4, t_5\}, (Choice(\mathcal{E}, \mathcal{S}, \mathcal{B}), g()))$ that returns t_1, followed by $next([t_3, t_1], \{t_1, t_2, t_3, t_4, t_5\}, (Choice(\mathcal{E}, \mathcal{S}, \mathcal{B}), g()))$ that returns $\{t_2, t_5\}$.

Formally, given a test suite T, a set of criteria $\{c_1, c_2, c_3, \ldots, c_m\}$, and a function $\mathcal{H}()$, our formulation of *Choice* may be represented as a function $\mathcal{C}(T, \{c_1, c_2, c_3, \ldots, c_m\}, \mathcal{H}())$ that returns a set/sequence of test cases $\{t_1, t_2, t_3, \ldots, t_n\}$ satisfying the following properties:

1. $t_i \in T$, for all $(1 \leq i \leq n)$.
2. All tests are ordered/selected by a single criterion c_x, $(1 \leq x \leq m)$, the one that performed the best as determined by $\mathcal{H}()$.

Note that if the merging function $\mathcal{G}()$ in our formal treatment of the merge hybrid is carefully constructed as a selection function that simply drops all of the nonselected (nonchosen) criteria, then the choice hybrid may, in principle, be viewed as a special case of merge. However, because the spirit of our choice formulation is to consider only a single criterion and that of merge is to consider all criteria, we discuss them separately.

3.3 Representing Higher Order Criteria

We now combine *Rank*, *Choice*, and *Merge* to form higher order criteria. We first motivate the need for higher order criteria. Consider our example of *Merge* with $\mathcal{S}$ and $\mathcal{B}$; recall that we had selected T_3 first, then had a tie between T_2 and T_4; finally, we added the remaining tests in any order. This left us with a less than satisfactory solution, where more than half of the test cases were ordered arbitrarily. To help remedy this situation, let us say that we wanted to add a secondary criterion, $\mathcal{E}$, that would be used as a tie breaker as well as to order the remaining test cases. We could come up with something like $Rank((Merge((\mathcal{S}, \mathcal{B}), g(); AF)), (\mathcal{E}, g()))$. Our overall call would look like:

```
next(
    [],
    {T1, T2, T3, T4, T5},
    Rank(
        (Merge((S, B), g(); AF)),
        (E, g())
    )
).
```

That is, our first criterion is a *Merge* hybrid of statement and branch coverage. If this hybrid fails to order the test cases, we use event coverage as a tie breaker, which is why we *Rank* these criteria. To execute our higher order criterion, we design our invocation function to use the $Rank_{H(c_x,c_y,\ldots,c_z)}$ invocation function shown in Fig. 3. The first element in the `Rank` specification is extracted (Line 5). Instead of being a base case, as we have seen in earlier examples, it is a second-order criterion *Merge*. This criterion is used in a *next()* call (Line 8), which expands to $next([], \{T_1, T_2, T_3, T_4, T_5\}, Merge((\mathcal{S}, \mathcal{B}), g(); AF))$, which turns out to be a call that we have seen handled in *Merge* earlier. Its output is T_3. The next invocation $next([T_3], \{T_1, T_2, T_3, T_4, T_5\}, Merge((\mathcal{S}, \mathcal{B}), g(); AF))$ returns $\{T_2, T_4\}$, which is a tie, to be handled by *Rank*. In the code for $Rank_H$, e now becomes $(\mathcal{E}, g())$ (Line 13), s is $\{T_2, T_4\}$ (Line 17), and the matrix $\mathcal{E}$ is reduced to rows corresponding to T_3, T_2, and T_4 (Line 18); let us denote this matrix by $\mathcal{E}[2, 3, 4]$. The subsequent invocation of *next()* (Line 8) expands to $next([T_3], \{T_2, T_4\}, (\mathcal{E}[2, 3, 4], g()))$, which returns $\{T_2, T_4\}$ because both T_2 and T_4 cover event e_3, not covered by T_3. In this case, event coverage was not a good tie breaker.

This foregoing example illustrates that we can recursively compute the *next()* functions by using the tree-like specification that combines *Rank*, *Merge*, and *Choice*.

4 Demonstrations

We demonstrate the usefulness of our representation in three ways. First, in Section 4.1, we qualitatively show the usefulness of our formulation and operators for the hybrid problem by *recasting others' previous uses of multiple criteria* in terms of our formulation and three operators. Second, in Section 4.2, we qualitatively show the usefulness of our formulation and operators for the hybrid problem by *developing new hybrid criteria*; we reuse results from our previous empirical study to understand the performance of individual criteria and combine some of them [3]; at the same time, we demonstrate how one would formulate hybrid criteria in practice. Finally, in Section 4.3, we empirically evaluate the effectiveness of hybrid-criteria prioritization by *empirically evaluating the new criteria*; additional evidence of this has been provided by other researchers who have used multiple criteria for test case prioritization.

4.1 Recasting Previous Work

Because we want our demonstration to be as complete as possible, we did an extensive systematic-mapping-like search to find others' previous work on hybrid criteria for performing regression testing tasks. We started with a search of the digital libraries, defined inclusion/exclusion criteria, performed focused targeted examination of likely researchers in the field, and finally identified 44 papers. More specifically, we searched the IEEE and ACM digital libraries for the terms "test case prioritization," "test suite prioritization," "test suite reduction," "test case reduction," "test suite minimization," "test case minimization," "test case selection," and "test suite selection." We merged all the results to eliminate overlaps; this gave us 593 papers. Because we did not want to compromise on the quality of

our search, we manually examined each paper. We manually excluded all papers that did not use hybrid. Finally, after a resource-intensive process of elimination, we were left with 44 papers that we studied in great detail. Of these 44 papers, we now recast work of Walcott et al. [63], Yoo and Harman [68], Harrold et al. [21], Jeffrey et al. [30], Sampath et al. [52], Lin et al. [36], Hsu and Orso [25], Black et al. [2], and Mirarab et al. [41] in terms of our new representation. Finally, we also recast our own previous work [3], in which we developed a hybrid criterion for test case prioritization. Because all this work has already been summarized in Section 2, we mention parts that are relevant to our representation.

Walcott et al. [63] first select tuples of test cases based on execution time and then use these tuples as the first generation of solutions for a genetic algorithm that evaluates the fitness of these tuples with respect to code coverage. Since they apply the criteria in series, their approach would fall under our classification of *Rank*. Though in their paper they present a single genetic algorithm that uses two criteria, since the criteria are applied in series (for the interested reader, in their Fig. 2, lines 1 through 5 represent selecting test cases based on execution time, and lines 6 through 21 correspond to selecting test cases based on code coverage), we treat their algorithm as having two $g()$ functions. In particular, the first invocation of $next()$ could be written as

$$next([], \{T_1, \ldots, T_i, \ldots, T_n\}, \\ Rank((ExecT, g_{ExecT}()), (C, g_c())),$$

where the function $g_{ExecT}()$ is a function that selects tuples based on execution time, $ExecT$ is the vector that contains execution time of each test case, function $g_c()$ is the genetic algorithm that Walcott et al. apply to select the prioritized test order, and C is the matrix containing the mapping between the test cases and the code covered. The $g_{ExecT}()$ function selects test tuples from the test suite that can be executed within a predetermined maximum execution time. The genetic algorithm in the $g_c()$ function uses a fitness function that assigns fitness values to each test tuple based the percentage of program code covered by the tuple and the time at which each test case covers its associated program code. The algorithm also provides for applying crossover and mutation to create new hybrid test tuples. In one set of experiments, they use method coverage and in another set of experiments, they use block coverage. In the subsequent invocation of *next()*, test cases that are selected based on completing execution within a predetermined maximum time are passed to the genetic algorithm, g_c, for ordering.

Yoo and Harman [68] present a hybrid algorithm for Pareto-efficient test suite reduction is an implementation of *Merge*. In particular, the first invocation of the $next()$ function could be written as

$$next([], \{T_1, \ldots, T_i, \ldots, T_n\}, \\ Merge((ExecT, C), g_{hybrid}())),$$

where $ExecT$ refers to the mapping between test cases and execution time, C refers to a mapping between code coverage and test cases, the g_{hybrid} function in their approach uses an additional-greedy algorithm and a genetic algorithm (NGSA-II). In the subsequent invocation of *next()*, again, all the criteria are considered together to select the next test to add to the test order.

Harrold et al. [21] propose three ways in which multiple selection criteria can be applied for test suite reduction. The first case they propose is when test cases, TS_1, from criterion, C_1, are a subset of test cases, TS_2, from another criterion, C_2. In this case, they propose applying the reduction algorithm on TS_1 to obtain a reduced suite. Then, they propose using this reduced suite as the initial reduced suite and applying the reduction technique on the test cases from TS_2 that cover requirements beyond those covered by the initial reduced suite. A similar approach is followed when the two test sets TS_1 and TS_2 have some overlap but one is not a subset of the other. In this case, first the reduction algorithm is applied on TS_1, then the intersection between the reduced suite and TS_2 is used as the initial reduced suite when applying the algorithm on TS_2. The third situation they propose is when TS_1 and TS_2 do not have any common elements; however, what they propose in this case is not an application of multiple criteria since the reduced suites are computed independently for the two test suites. For the first and second case, since the reduction algorithm is applied in series on the two test sets obtained from two criteria, the hybrid approaches of Harrold et al. [21] are an application of our *Rank* approach. The first invocation of $next()$ could be written as

$$next([], \{T_1, \ldots, T_i, \ldots, T_n\}, \\ Rank((C_1, g_{hgs}()), (C_2, g_{hgs}())),$$

where C_1 and C_2 are the two criteria that are used during test suite reduction and the $g_{hgs}()$ function refers to the HGS reduction algorithm proposed in their work. In the subsequent invocation of *next()*, the reduction algorithm will be applied on the tests from C_2 that cover requirements beyond the reduced suite selected by applying C_1.

Jeffrey et al. [30] use two or more criteria in their test suite reduction approach called *reduction with selective redundancy*. When a test case is deemed redundant by one criterion, they study the requirement coverage of the test with respect to a second criterion. If the test covers some uncovered requirements of the second criterion, the test is added to the reduced test suite. Since the criteria are applied in series, Jeffrey et al.'s approach can be viewed as an implementation of our *Rank* hybrid formalization. In their experiments, they use branch coverage, all-uses coverage, and subpaths of length 3 coverage. The first invocation of the $next()$ function, in this case, would be

$$next([], \{T_1, \ldots, T_i, \ldots, T_n\}, \\ Rank((B, g_{rsr}()), (U, g_{rsr}()), (SP, g_{rsr}())),$$

where B, U, and SP are the mappings between test cases and branch, all-uses, and subpaths of length 3 criteria, respectively; g_{rsr} is the selective redundancy algorithm proposed in their work. In the subsequent invocation of *next()*, test cases that are redundant by branch coverage will be evaluated with respect to all-uses coverage.

Sampath et al. [52] propose an application of both of our *Merge* and *Rank* formalizations. We discuss the *Rank* hybrid formalization here. They apply the two sets of criteria in series. In their experiments, they use method coverage as the program coverage-based criterion. They use usage-based

criteria derived for web applications, such as base requests, base requests and name, base requests and name-value pairs, sequences of base requests of size 2, and sequences of base requests and name of size 2. The first invocation of the *next()* function in this case would be

$$next([], \{T_1, \ldots, T_i, \ldots, T_n\}, \\ Rank((M, g_{mod-hgs}()), (base, g_{mod-hgs}())),$$

where the literals M and $base$ refer to the mapping between the test cases and methods, and base requests, respectively. The function $g_{mod-hgs}$ refers to their implementation of the modified HGS algorithm [21]. Similar invocations can be written when method coverage is combined with the other usage-based criteria. A subsequent invocation of *next()* would take the test cases that are tied with respect to method coverage and evaluate them for breaking ties with respect to base requests coverage.

Lin et al. [36]'s approach is an application of our *Rank* hybrid formalization. In their experiments, they use branch coverage as the primary criterion and def-use pair coverage as the secondary criterion. The first invocation of the *next()* function in this case would be

$$next([], \{T_1, \ldots, T_i, \ldots, T_n\}, \\ Rank((B, g_{m-hgs}()), (DU, g_{m-hgs}())),$$

where the B and DU are the mappings between the test cases and the branches, def-use pairs that are covered, respectively. The function g_{m-hgs} refers to the modified-HGS algorithm that they implement which incorporates tie-breaking into Harrold et al.'s [21] reduction algorithm. They also augment another reduction algorithm, the GRE algorithm [5], [6] with their tie breaking mechanism, so another $g()$ function that they use is the $g_{m-gre}()$ function. In a subsequent invocation of *next()*, test cases that are tied with respect to branch coverage will be evaluated with respect to def-use coverage to break the ties.

Hsu and Orso [25] encode multicriteria test suite reduction as a binary integer linear programming problem. They propose three policies based on which criteria can be combined. They consider two types of criteria, absolute and relative criteria. An absolute criterion (e.g., maintain branch coverage) introduces a constraint on the reduction, while a relative criterion (e.g., minimize test execution time) presents an objective. They propose three ways in which criteria can be combined, namely, weighted policy, prioritized policy, and hybrid policy. Note that their policies are applied only to the relative criteria or objectives.

In the *weighted* policy, a weight is given to each objective and all the objectives are considered at once during test suite reduction. Their *weighted* policy is an implementation of our *Merge* formalization. In their experiments, they use the absolute criterion, "maintain code coverage," and the relative criteria, "minimize number of test cases," "minimize execution time," "maximize number of error revealing test cases." The next() function invocation for their *weighted* approach can be written as

$$next([], \{T_1, \ldots, T_i, \ldots, T_n\}, \\ Merge((C, NumTests, ExecTime, ErrorTests), \\ g_{mints}())),$$

where $C, ExecTime, ErrorTests$ correspond to the mapping between test cases and the criteria, code coverage, execution time and errors revealed. *NumTests* is a vector representing number of test cases. Function g_{mints} refers to their reduction algorithm that encodes the problem as an ILP problem and feeds to several ILP solvers. In a subsequent invocation of *next()*, all the criteria are again used together to determine the next test case for the minimized test suite.

On the other hand, their *prioritized* policy assigns priorities to each objective and selects them one at a time in combination with the absolute criterion to perform the test suite reduction. Therefore, their *prioritized* policy is an implementation of a higher order hybrid, where a *Merge* is performed between the absolute criterion and the relative criteria, upon which a *Rank* formalization is applied to handle the priorities. The *next()* function invocation for their weighted approach can be written as

$$next([], \{T_1, \ldots, T_i, \ldots, T_n\}, \\ Rank(\\ Merge((C, NumTests), g_{mints}), \\ Merge((C, ExecTime), g_{mints}), \\ Merge((C, ErrorTests), g_{mints}) \\ )).$$

Finally, their *hybrid* policy divides objectives into groups and assigns priorities to groups. Weights are assigned to objectives within the group. Their weighted policy is thus a higher order hybrid of our *Union* and *Rank* approaches. They do not experimentally evaluate the *weighted* policy. To illustrate the formalization of their hybrid policy, here we consider a situation where the relative criterion "minimize number of test cases" is in one group, and the relative criteria "minimize execution time" and "maximize number of error revealing test cases" are in another group, and the first group has priority over the second group. Note that the absolute criterion must be used in combination with each group of relative criteria when minimizing the test suite. The *next()* function invocation for their *hybrid* policy can be written as

$$next([], \{T_1, \ldots, T_i, \ldots, T_n\}, \\ Rank(\\ (Merge((C, NumTests), g_{mints}())) \\ (Merge((C, ExecTime, ErrorTests), g_{mints}())) \\ ) \\ ).$$

Black et al. [2] represent the test suite reduction problem using a binary ILP representation. They propose a single objective model that minimizes a test suite based on def-use association coverage; however, they note that if two or more test cases cover the same set of def-use associations, only one test is selected. To overcome the risk of removing an error-revealing test case, they also present a bi-objective model that minimizes a test suite based on def-use coverage and the ability of a test case to reveal an error. They note that any other criteria can also be used in the models. The problem can be formulated such that one part of the objective can take precedence over the other using a weight between 0 and 1. The value of the weight determines to what extent each of the objectives contribute to the composite objective function. Since their goal is to satisfy

TABLE 2
Test Case Prioritization Criteria

No.	Criterion	Coverage Elements
1	*1-way*	Number of previously uncovered parameter-values
2	*2-way*	Number of previously uncovered 2-way interactions between parameter-values
3	*PV-LtoS*	Number of parameter-values (descending)
4	*PV-StoL*	Number of parameter-values (ascending)
5	*Action-StoL*	Number of windows/requests (ascending)
6	*Action-LtoS*	Number of windows/requests (descending)
7	*UniqWin*	Number of unique windows/requests (descending)
8	*MFPS*	Number of most frequently present windows/requests sequence of size 2 (descending)
9	*Weighted-Freq*	The number of present windows/requests sequences of size 2 which is scaled by weights for each sequence based on the number of times that a sequence appears in the test suite (descending)

both the criteria at the same time, their approach is an application of our *Merge* approach. The *next()* function invocation for their approach can be written as

$$next([], \{T_1, \ldots, T_i, \ldots, T_n\}, \\ Merge((Uses, ErrorTests), g_{ilp}())),$$

where *Uses* and *ErrorTests* refers to the mapping between test cases and the criteria, all-uses, and error revealing ability of the test, respectively. The function g_{ilp} refers to the algorithms that were used to compute the minimized test suite using integer linear programming. A subsequent invocation of *next()* will select test cases with the goal of satisfying both all-uses and error-revealing-ability criteria.

Mirarab et al. [41] use two criteria to perform multicriteria test case selection. They place limits on the number of test cases that can be selected. The two criteria they use are code coverage-based criteria, which are fed to an ILP solver that selects the final test suite. Since the goal is to satisfy both the criteria, their approach is an implementation of *Merge*. Specifically, the *next()* function invocation for their approach can be written as

$$next([], \{T_1, \ldots, T_i, \ldots, T_n\}, \\ Merge((D_{sum}, D_{min}), g_{ilp}())),$$

where D_{sum} and D_{min} are their code coverage-based criteria. The D_{sum} criteria seeks to maximize coverage of all program elements, and the D_{min} criteria seeks to maximize the minimum coverage across all program elements. The function g_{ilp} refers to the ILP solver that produces the final test suite. A subsequent invocation of *next()* will select test cases with the goal of satisfying both the D_{sum} and D_{min} criteria.

In our previous work [3], we proposed a hybrid approach based on a schedule in which we prioritize by one criteria until the first 10 percent block of test cases is encountered with no increase in effectiveness, and then switch to a second criterion. Since we are applying the criteria in series and switching from the first to second after observing no improvement in effectiveness from the first, our hybrid approach is an implementation of *Rank*. We use a frequency criterion as the first criterion, and a combinatorial criterion, 2way, as the second criterion. The next() function invocation can be written as

$$next([], \{T_1, \ldots, T_i, \ldots, T_n\}, \\ Rank((Freq, g_{schedule}()), (2way, g_{schedule}())),$$

where *Freq* and $2way$ represent the mapping between test cases and the frequency and 2way coverage, while the function $g_{schedule}$ refers to the schedule-based algorithm based on which we select test cases for prioritization. A subsequent invocation of *next()* will use the 2way criterion, when a 10 percent block of tests are selected by the frequency criterion that do not contribute to increase in fault detection effectiveness.

We see that we were able to successfully recast all the above work in terms of *Merge* or *Rank* to combine criteria; we did not need to use *Choice*. We have not presented a detailed analysis of the 44 papers that employ hybrid criteria because of space reasons. However, we have studied all 44 papers in great detail and conclude that 15 use *Rank* [3], [10], [14], [19], [21], [24], [28], [30], [34], [36], [52], [63], [70], [72], [73], 27 use *Merge* [1], [2], [4], [7], [8], [11], [17], [18], [23], [25], [26], [27], [32], [35], [37], [38], [40], [41], [42], [44], [45], [47], [54], [60], [66], [67], [68], but none use *Choice*. It may be that no prior studies examine *Choice* because it is expensive under certain circumstances. For instance, recomputing the data based on all of the metrics for every criteria at each step may be expensive. Moreover, two papers use a higher order hybrid of *Merge* followed by *Rank* [61], [64]. We believe that this is a positive result because although our three operators were not directly influenced by the previous papers (indeed, we first created the formulations and later recast others' work in terms of the three operators), we were still successful in recasting all others' previous work. In future work, we will examine why the *Choice* hybrid has never been used.

4.2 Developing New Hybrid Criteria

4.2.1 Background

In previous work [3], we develop and empirically evaluate several criteria for test case prioritization. In this section, we use the artifacts from this previous work to experiment with hybrid techniques. Due to space constraints, we do not provide all details of the previous results. However, in an effort to make this paper self-contained, we provide sufficient details needed for this current work. The interested reader is referred to past reported work for full details [3].

We list some of the criteria in Table 2. Because we deal with web and GUI applications, our criteria consider GUI *windows* and web GET/POST *requests*. For web applications, we model an event as a POST or GET request together with its set of *parameters* and their *values*. For a GUI application, we model an event as the execution of a *termination* user action, such as `OK` or `Cancel` together with the settings of any other widgets in the window, such as the status of a check-box, selected radio button from a group of radio

TABLE 3
Composition of the Applications and Test Suites in Our Study

	Calc	Paint	SSheet	Word	Book	CPM	Masplas
Application Type	GUI	GUI	GUI	GUI	Web	Web	Web
Programming Language(s)	Java	Java	Java	Java	JSP, HTML MySQL	Java servlets, HTML File-based datastore	Java servlets, HTML MySQL
Windows	2	11	9	12	9	65	18
Parameter-values	85	247	188	156	415	4166	646
LOC	9916	18376	12791	4893	7615	9401	999
Classes	141	219	125	104	11	75	9
Methods	446	644	579	236	319	173	22
Branches	1306	1277	1521	452	1720	1260	108
Total no. of tests	300	300	300	250	125	890	169
Largest count of actions in a test case	47	51	50	50	160	585	69
Average count of actions in a test case	14.5	19.7	19	27.8	29	14	7
No. of seeded faults	175	182	79	96	40	135	29
Min. no. faults found by a test	0	0	0	0	6	0	1
Avg. no. faults found by a test	9.4	1.6	4	24	21.43	4.67	4.62
Max. no. faults found by a test	48	64	71	87	32	33	15

buttons, the text string in a text-field; the widgets that can be assigned such settings form the *parameters* and the particular settings form the *values*.

We applied these criteria on test suites of seven event-driven systems, four GUI and three web applications. Applications *Calc*, *Paint*, *SSheet*, and *Word* are GUI applications, while *Book*, *CPM*, and *Masplas* are web-based applications. Table 3 shows the main characteristics of the subject applications, test cases, and fault matrices. The applications range in size from 1,000 to 19,000 lines of code. They contain a large number of windows and parameter values. The test cases for the GUI applications (between 250 to 300 tests) are generated from a model of the GUI. Web application usage logs are converted into test cases for the web applications [53]. Faults are manually seeded into the applications by graduate and undergraduate students who were familiar with Java, Servlets, and JavaServer Pages (JSP) [53], [57]. The seeded faults fall under five categories, namely, data store faults (faults that exercise application code that interacts with the data store), logic faults (faults that are introduced due to logic errors in application code), appearance faults (faults in application code that alter the way the user views the page), link faults (faults that change the location pointed by a hyperlink), and form faults (faults that exist in name-value pairs and actions in forms). The number of faults varies in each application, from 29 to 182 faults.

4.2.2 Creating the Hybrids

The criteria of Table 2 serve as individual criteria for this current work. Because we have empirically evaluated them in previous work, this gives us some intuition on how to combine them to create the hybrids. (We discuss this intuition in Section 4.2.3.) Specifically, we define five new criteria:

- $Merge_{U(2-way,1-way)}$ combines *2-way* and *1-way*, giving both equal weight.
- $Merge_{U(2-way:2,1-way:1)}$ combines *2-way* and *1-way*, such that *2-way* gets twice the weight of *1-way*.
- $Merge_{U(2-way:1,1-way:2)}$ combines *2-way* and *1-way*, such that *1-way* gets twice the weight of *2-way*.
- $Rank_{H(2-way,PV-LtoS,Action-LtoS,UniqWin)}$ combines *2-way*, *PV-LtoS*, *Action-LtoS*, and *UniqWin* such that *2-way* is used as the primary criterion; only when it is tied, *PV-LtoS* is used to break the tie; the tie-breaking chain continues to *Action-LtoS* and *UniqWin*.
- $Choice_{T(PV-StoL,Action-StoL,UniqWin,MFPS)}$ combines *PV-StoL*, *Action-StoL*, *UniqWin*, *MFPS* such that only one is applied at any time based on a selection criterion.
- Higher order hybrid criteria that first apply $Merge_{U(2-way,1-way)}$ and then use *PV-LtoS*, *Action-LtoS*, and *UniqWin* to break ties. Similarly, first $Merge_{U(2-way:2,1-way:1)}$ and $Merge_{U(2-way:1,1-way:2)}$ are applied and the other three criteria are used to break ties.

We implemented the different criteria in C++ and Python. A script then calls the individual methods to compute the scores based on the criteria. We now describe each hybrid in detail, starting with our intuition for creating each.

4.2.3 $Merge_{U(2-way,1-way)}$, $Merge_{U(2-way:2,1-way:1)}$, and $Merge_{U(2-way:1,1-way:2)}$

In our past work, we observed that *2-way* was consistently among our top three prioritization criteria. The *2-way* criterion selects test cases that cover the most uncovered 2-way parameter-value interactions between windows. On the other hand, *1-way* sometimes outperformed the *2-way* criterion. We hypothesize that while covering all interwindow event interactions is valuable that covering all events early is also important and that we should examine this further. We thus combine *1-way* and *2-way* into a hybrid criterion. By considering both *1-way* and *2-way*, in some cases, we give priority to variety of windows covered (as selected by *1-way*) in addition the pairwise interactions between windows (as selected by *2-way*). *Merge* is our most natural choice for this combination because it simultaneously considers the criteria—it gives importance to a test case that covers the maximum number of previously uncovered pairs of events *and* the maximum number of events. This leads us to our first hybrid criterion, $Merge_{U(2-way,1-way)}$. The first invocation of the $next()$ function for this hybrid could be written as

$$next([], \{T_1, \ldots, T_i, \ldots, T_n\}, \\ Merge((M_{1way}, M_{2way}), g_{greedy}())),$$

where M_{1way} refers to the mapping between test cases and the 1-way interactions they cover, M_{2way} refers to a mapping between test cases and the 2-way interactions they cover, the g_{greedy} function is our greedy prioritization algorithm.

However, because *2-way* is our best performing criterion, we would not like *1-way* to "drag it down." Hence, we give *2-way* more importance by assigning it a higher weight; we assign the weight 2 to *2-way* and 1 to *1-way*; this leads us to our second criterion, $Merge_{U(2-way:2,1-way:1)}$. We also observe that the number of criteria elements for *2-way* far exceed those for *1-way*. This is because *2-way* considers *pairs* of parameter values, whereas *1-way* considers only individual parameter values. If we have n parameter values, then *2-way* may, in principle, need to consider $\binom{n}{k}$ pairs. By combining *2-way* with *1-way*, we run the risk of "overwhelming" *1-way* because of the much larger influence of *2-way*. This is why we develop our third criterion—$Merge_{U(2-way:1,1-way:2)}$ that assigns the weight 1 to *2-way* and 2 to *1-way*.

Merge, as used above, takes matrix representations of the individual criteria and performs a horizontal concatenation, with weights if needed, to obtain a single matrix for prioritization.

4.2.4 $Rank_{H(2-way,PV-LtoS,Action-LtoS,UniqWin)}$

In our past work, we also observed that *2-way* led to a large number of ties between test cases. In such cases, we were selecting one of the tied test cases by random selection. We hypothesize that the results may be better if we used one or more secondary criteria to break ties as done by [36], [52].

We now combine *2-way* with *PV-LtoS*, *Action-LtoS*, and *UniqWin*. Criterion *PV-LtoS* orders test cases in descending order of the number of parameter values in the tests, *Action-LtoS* orders test cases in descending order of the number of windows, and *UniqWin* orders tests in descending order of number of unique windows covered by the test. The criteria *PV-LtoS* and *Action-LtoS* are among the top three best criteria in several of our subject applications. For example, *PV-LtoS* is among the top three best criteria in four out of the seven subject applications, *Action-LtoS* in one out seven. The criterion *UniqWin* is also in the top three in one out of the seven applications, though it performs poorly in other applications.

Our hybrid formulation is based on *Rank*, where *2-way* is the primary criterion and *PV-LtoS*, *Action-LtoS*, and *UniqWin* are used as tie breakers, in that order. The first invocation of $next()$ for this hybrid could be written as

$$next([], \{T_1, \ldots, T_i, \ldots, T_n\}, \\ Rank((M_{2way}, g_{greedy}()), (M_{PV\text{-}LtoS}, g_{greedy}()), \\ (M_{Action\text{-}LtoS}, g_{greedy}()), (M_{UniqWin}, g_{greedy}())),$$

where M_{2way}, $M_{PV\text{-}LtoS}$, $M_{Action\text{-}LtoS}$, $M_{UniqWin\text{-}LtoS}$ refers to the mapping between the test cases and 2-way interactions they cover, the number of parameter values, number of actions, and number of unique windows they cover, respectively. The function $g_{greedy}()$ refers our greedy prioritization algorithm.

4.2.5 $Choice_{T(PV\text{-}StoL,Action\text{-}StoL,UniqWin,MFPS)}$

Our past results also showed that the criteria *PV-StoL*, *Action-StoL*, *UniqWin*, and *MFPS* individually did poorly. We hypothesize that these criteria are useful when ordering parts of the test suite; however, when applied individually across the entire test suite, they are unable to perform. This is because the criteria individually cover characteristics of the source code that do not contribute to increase in fault detection effectiveness, for example, *PV-StoL* gives priority to tests that cover small number of parameter values, but covering few parameter values is likely to translate to a poor code coverage. However, by using a combination of the criteria, different characteristics are covered as the test suite is ordered, thus cumulatively resulting in an increase in fault detection effectiveness.

We now combine *PV-StoL*, *Action-StoL*, *UniqWin*, and *MFPS* in such a way that each is used only on a small (10 percent) part of the test suite at a time. This naturally leads to an application of *Choice*. In our instantiation of *Choice*, which we call $Choice_{(PV-StoL,Action-StoL,MFPS)}$, we use a schedule-based approach. We select the most effective test order every 10 percent of the test suite. We hypothesize that as tests are selected, the remaining tests could be stronger with respect to a different criterion and should thus be evaluated under the auspices of other criteria. The goodness of every 10 percent of the test suite is determined by the APFD value of the partial test suite.

The first invocation of $next()$ for this hybrid could be written as

$$next([], \{T_1, \ldots, T_i, \ldots, T_n\}, \\ Choice((M_{PV\text{-}StoL}, M_{Action\text{-}StoL}, \\ M_{UniqWin}, M_{MFAS}), g_{greedy}()),$$

where $M_{PV\text{-}StoL}$, $M_{Action\text{-}StoL}$, $M_{UniqWin}$, M_{MFAS} refer to the mapping between the test cases and the number of parameter-values covered, number of actions covered, number of unique windows covered, and number of times the most frequently present sequence is covered, respectively, and the $g_{greedy}()$ function refers our greedy prioritization algorithm.

4.2.6 Higher Order Hybrid Criteria

In our pilot studies of the above hybrid formulations, we noticed that in many instances criteria $Merge_{U(2-way,1-way)}$, $Merge_{U(2-way:2,1-way:1)}$, and $Merge_{U(2-way:1,1-way:2)}$ create a large number of ties. In such situations, instead of breaking ties at random, we use *PV-LtoS*, *Action-LtoS*, *UniqWin*, in that order, to break the ties, thus, creating what are effectively second-order hybrid criteria similar to that discussed in Section 3.2. The formulation is one of *Rank*, with the first criterion itself being a *Merge* hybrid of *2-way* and *1-way*, weighted appropriately. For $Merge_{U(2-way,1-way)}$, we obtain $Rank_{H(Merge_{U(2-way,1-way)},PV-LtoS,Action-LtoS,UniqWin)}$, i.e., if the criterion $Merge_{U(2-way,1-way)}$ produces ties, then the remaining criteria are used to break ties. Similarly, for $Merge_{U(2-way:2,1-way:1)}$, we obtain

$$Rank_{H(Merge_{U(2-way:2,1-way:1)},PV-LtoS,Action-LtoS,UniqWin)},$$

and for $Merge_{U(2-way:1,1-way:2)}$, we obtain

$$Rank_{H(Merge_{U(2-way:1,1-way:2)},PV-LtoS,Action-LtoS,UniqWin)}.$$

4.3 Stand-Alone versus Hybrid

We now study the effectiveness of the hybrid criteria that we created in Section 4.2. Other researchers [2], [25], [30],

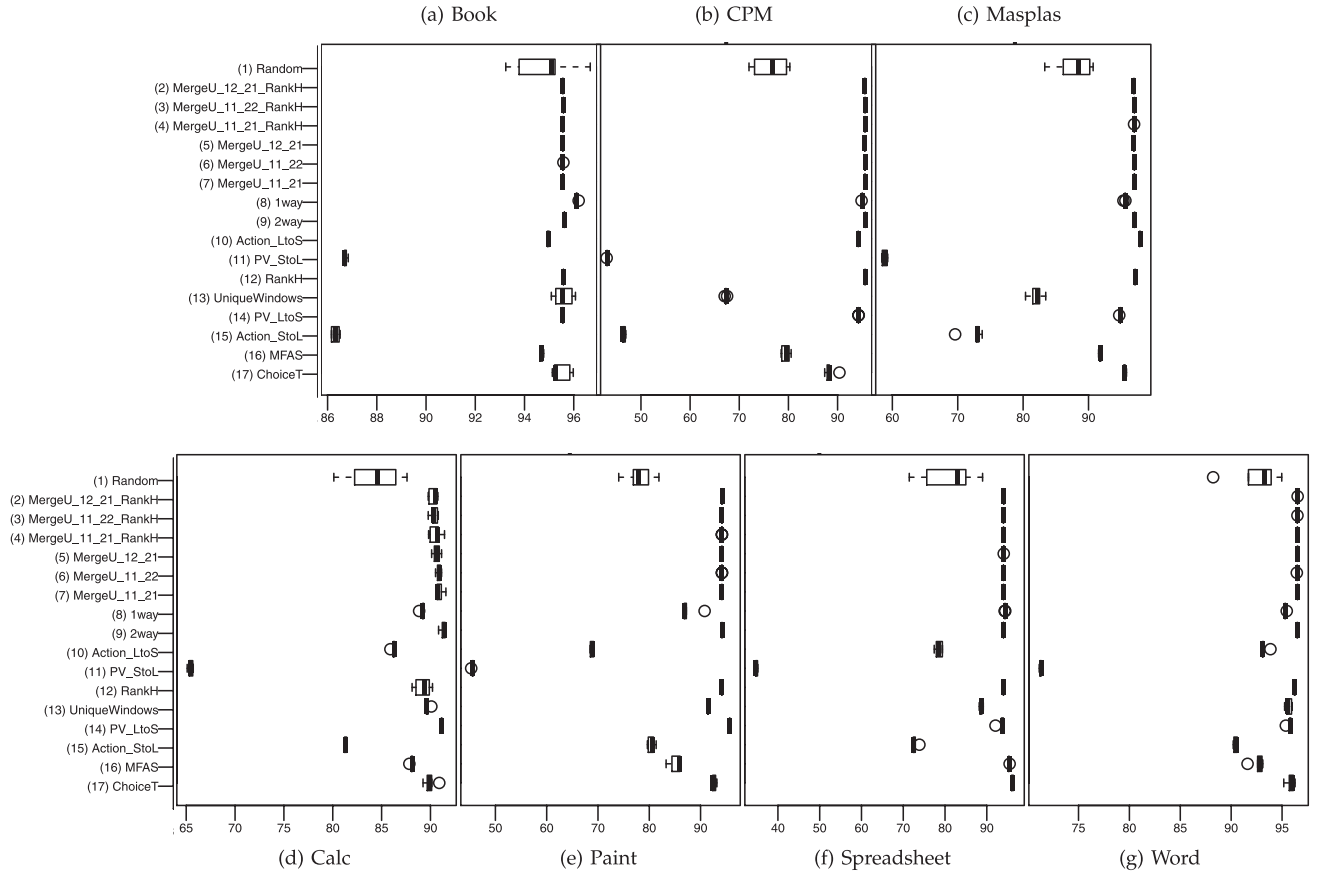

Fig. 4. Stand-alone versus hybrid results. x-axis shows APFD values; the y-axis shows criteria.

[36], [52], [63] have shown the benefits of using hybrid criteria. Our goal in this section is not to show that hybrids work significantly better than nonhybrids, but, to provide closure to the hybrids we formulated in Section 4.2 by providing an example of how to empirically measure and evaluate them.

4.3.1 Metrics

We want to determine how the hybrid test orders compare with nonhybrid test orders. To measure effectiveness of a test order, we use the *average percent of faults detected* metric [49]. Although several metrics exist to evaluate prioritized test orders [9], [13], [46], [51], APFD is the most commonly used metric. As defined by Rothermel et al. [49], for a test suite T with n test cases, if F is a set of m faults detected by T, then let TF_i be the position of the first test case t in T', where T' is an ordering of T, that detects fault i. Then, the APFD metric for T' is given as

$$APFD = 1 - \frac{TF_1 + TF_2 + TF_3 + \cdots + TF_m}{mn} + \frac{1}{2n}. \quad (1)$$

Informally, APFD measures the area under the curve that plots test suite fraction and the number of faults detected by the prioritized test case order.

4.3.2 Methodology

We prioritize our given test cases using all hybrid and stand-alone criteria, and calculate the APFD of each test order. Because the prioritization algorithm uses randomness to order remaining test cases (those that are "left over" after all coverage requirements are satisfied), we repeat the prioritization five times so that the effect of randomness is reduced in our results.

We report the results using box plots shown in Figs. 4a, 4b, 4c, 4d, 4e, 4f, and 4g. A box plot is a concise representation of multiple data distributions; each distribution is shown as one box. The left and right edges of each box mark the third and first quartiles, respectively. The line inside the box marks the median value (it sometimes overlaps with the first/third quartile). The whiskers extend from the quartiles and cover 90 percent of the distribution. The remaining data-points (10 percent) are considered outliers and are shown as small circles beyond the whiskers. The box plots show the prioritization criterion on the y-axis, and their APFD values on the x-axis.

As observed in the box plots, the 5 runs produced the same APFD result, creating flat box plots. This is because even though the randomness creates different test orders, the tests that are involved in the randomness are not different in terms of finding additional faults to cause a change in the APFD. Thus, the final APFD values do not differ much, which leads to the flattened box plots seen in the graphs.

In addition, to compare all pairs of distributions, we conduct a statistical analysis. On conducting the F-test [43], we found that the variances of the distributions are not equal. Therefore, we perform Welch's t-test [43] on the APFDs of the five test orders for each pair of techniques to determine if two techniques that derive the test order are significantly different in their means. The null hypothesis in

TABLE 4
p-Values of t-Test Applied to Criteria

#	Criteria Pair	book	calc	cpm	masplas	paint	ssheet	word
1	1way, MergeU-11-21	0.00	0.00	0.00	0.00	0.00	0.00	0.00
2	2way, MergeU-11-21	-NA-	0.14	0.00	0.01	0.24	0.18	0.21
3	1way, MergeU-11-22	0.00	0.00	0.00	0.00	0.00	0.00	0.00
4	2way, MergeU-11-22	0.00	0.03	0.00	0.01	0.35	-NA-	0.49
5	1way, MergeU-12-21	0.00	0.00	0.00	0.00	0.00	0.00	0.00
6	2way, MergeU-12-21	-NA-	0.02	0.00	0.00	0.22	0.00	0.21
7	Action-LtoS, RankH	0.00	0.00	0.00	0.00	0.00	0.00	0.00
8	PV-LtoS, RankH	-NA-	0.00	0.00	0.00	0.00	0.25	0.00
9	UniqueWindows, RankH	0.98	0.27	0.00	0.00	0.00	0.00	0.01
10	2way, RankH	-NA-	0.00	0.00	0.00	0.01	-NA-	0.00
11	PV-StoL, ChoiceT	0.00	0.00	0.00	0.00	0.00	0.00	0.00
12	Action-StoL, ChoiceT	0.00	0.00	0.00	0.00	0.00	0.00	0.00
13	MFAS, ChoiceT	0.01	0.00	0.00	0.00	0.00	0.00	0.00
14	UniqueWindows, ChoiceT	0.72	0.47	0.00	0.00	0.00	0.00	0.38
15	MergeU-11-21-RankH, RankH	-NA-	0.03	0.07	0.00	0.07	0.18	0.00
16	MergeU-11-22-RankH, RankH	-NA-	0.04	-NA-	0.00	0.01	-NA-	0.00
17	MergeU-12-21-RankH, RankH	-NA-	0.05	-NA-	0.00	0.00	0.00	0.00
18	MergeU-11-21-RankH, MergeU-11-21	-NA-	0.31	0.21	0.00	0.36	1.00	0.28
19	MergeU-11-22-RankH, MergeU-11-22	0.02	0.05	0.18	0.18	0.69	-NA-	0.35
20	MergeU-12-21-RankH, MergeU-12-21	-NA-	0.24	0.75	0.75	0.20	0.55	0.82

each comparison is that the means of the two distributed populations are equal. The p-values from the t-test are shown in Table 4. In the next section, we discuss the performance of each of the hybrid techniques when compared to the nonhybrids.

4.3.3 Results and Analysis

$Merge_U$ versus nonhybrid criteria. Here, we compare the effectiveness of $Merge_{U(2-way,1-way)}$, $Merge_{U(2-way:2,1-way:1)}$, $Merge_{U(2-way:1,1-way:2)}$ hybrid techniques against the two nonhybrids, *1-way* and *2-way*, that are used in the *Merge* algorithm. We apply weights to the value of the *1-way* and *2-way* coverage to develop the hybrid combinations. For instance, $Merge_{U(2-way:2,1-way:1)}$ gives a weight of 2 for *2-way* and a weight of 1 for *1-way;* that is, we multiple the number of *2-way* combinations are scaled to be counted twice while the number of *1-way* combinations are not scaled. The rows relevant to this comparison are rows 5-9 in Figs. 4a, 4b, 4c, 4d, 4e, 4f, and 4g. The notation MergeU_11_21 in the figures and Table 5 indicates $Merge_{U(2-way:2,1-way:1)}$, the notation MergeU_11_22 indicates $Merge_{U(2-way:2,1-way:1)}$, and the notation MergeU_12_21 indicates $Merge_{U(2-way:1,1-way:2)}$.

We see that in most applications, the variations of $Merge_U$ are comparable to *2-way*. In most cases, $Merge_U$ is better than *1-way*. For spreadsheet and book, where *1-way* is better than *2-way*, we find that the $Merge_U$ techniques perform poorer than *1-way*. The weights on the two criteria do not appear to have an impact on the effectiveness of the test orders.

The first six rows of Table 4 show the p-values of comparing the $Merge_U$ test orders with *1-way* and *2-way*. We use "NA" when the two distributions are constant. From the first six rows of Table 4, we note that there is a statistically significant difference between the means of the *Merge* test suites and the nonhybrids for the web applications (p-value < 0.05). For three out of the four GUI applications, for $Merge_{U(2-way,1-way)}$, $Merge_{U(2-way:2,1-way:1)}$, and $Merge_{U(2-way:1,1-way:2)}$ the difference between them and *2-way* is not significant.

For applications where the difference between the means of the APFD is around 0.1 percent (e.g., in Paint the pair (*2-way*, $Merge_{U(2-way,1-way)}$)) we find that the difference is not significant (i.e., p-value > 0.05).

$Choice_T$ versus nonhybrid criteria. In this section, we present the results of comparing the

$$Choice_{T(PV-StoL,Action-StoL,UniqWin,MFPS)}$$

hybrid test order with the nonhybrid test orders generated from the criteria *PV-StoL*, *Action-StoL*, *UniqWin*, and *MFPS*. Rows 13-17 are the relevant data from Figs. 4a, 4b, 4c, 4d, 4e, 4f, and 4g for this comparison. From the graphs, we see that the $Choice_{T(PV-StoL,Action-StoL,UniqWin,MFPS)}$ hybrid technique performs better than the individual nonhybrid criteria for all the subject applications.

From the rows 11-14 in Table 4, we see that in most cases there is a statistically significant difference between the means of the $Choice_{T(PV-StoL,Action-StoL,UniqWin,MFPS)}$ and the nonhybrid techniques of *PV-StoL*, *Action-StoL*, *UniqWin*, and *MFPS*. These results suggest that the $Choice_{T(PV-StoL,Action-StoL,UniqWin,MFPS)}$ hybrid test criterion can be used to create effective test orders when the tester has prioritization criteria that individually perform poorly.

$Rank_H$ versus nonhybrid criteria. Here, we discuss the effectiveness of $Rank_{H(2-way,PV-LtoS,Action-LtoS,UniqWin)}$ hybrid criterion when compared the nonhybrids of *2-way PV-LtoS Action-LtoS*, and *UniqWin*. In Figs. 4a, 4b, 4c, 4d, 4e, 4f, and 4g, rows 9-13 contain the data relevant to this comparison. We see that the $Rank_{H(2-way,PV-LtoS,Action-LtoS,UniqWin)}$ hybrid algorithm is comparable to *2-way* and in most cases is better than the other nonhybrid criteria. For Calc, however, $Rank_{H(2-way,PV-LtoS,Action-LtoS,UniqWin)}$ performs poorer than *2-way* by a small margin. This could be because Calc was one of our smaller applications with only two windows.

The means of *2-way* and

$$Rank_{H(2-way,PV-LtoS,Action-LtoS,UniqWin)}$$

as seen in row 10 in Table 4 show a statistically significant difference. Thus, the $Rank_{H(2-way,PV-LtoS,Action-LtoS,UniqWin)}$ hybrid technique can be used to generate effective test orders with the help of intelligent tie breaking.

Higher order hybrid criteria. For our higher order hybrid criteria, where we combine $Merge_U$ and $Rank_H$, we observe that the hybrids (rows 2, 3, 4) do not perform much better than the individual first order hybrid criteria (rows 5, 6, 7, 12 in Figs. 4a, 4b, 4c, 4d, 4e, 4f, and 4g). Correspondingly, from Table 4 rows 15 to 20, we see that several of the pairs do not show a statistically significant difference. We believe the performance of hybrid criteria in our experiments is an artifact of our dataset and the faults seeded in our subject applications. The $Merge_U$ criteria alone are very effective, such that we do not see an improvement on applying the higher order criteria.

Summary. In summary, we find that our hybrids perform better than the nonhybrid criteria in most of our subject applications. We note that the hybrids perform better than a random ordering of the tests (row 1 in graphs). Previous literature that we reviewed shows the benefits of a variety of $Merge$ and $Rank$ hybrid techniques. We also provide an example that explores the $Choice$ hybrid approach for the scenario when a tester has several poorly performing criteria from which they need to choose or combine. The formalizations that we have previously proposed allow the easy application of the individual criteria to create the hybrids that we study here.

We note that in our subject applications and experiments the APFD of the different hybrids is relatively high, so the distinction between the criteria is not obviously apparent.

TABLE 5
Summary of 44 Hybrid Papers We Studied

Reference	Keywords	Criteria used in hybrid
	Merge	
[68]	pareto efficiency, genetic algorithm variant	code coverage, execution time, fault detection history
[67]	pareto efficiency, genetic algorithm variant	code coverage, execution time, fault detection history
[25]	integer linear programming	code coverage, execution time, error-revealing test cases
[2]	integer linear programming	all-uses, error revealing ability
[41]	integer linear Programming	code coverage-based
[7]	pareto efficiency, genetic algorithm variant	code coverage, execution time
[27]	IR-based technique, latent semantic indexing traceability links between source code	code coverage, requirements-coverage execution cost, execution cost and requirements
[4]	agglomerative hierarchical clustering	code complexity, fault history
[11]	ant colony algorithm	test coverage, test cost
[8]	particle swarm optimization	requirements coverage, execution time
[37]	weighted sum/average	faults detection, changed module execution
[42]	genetic algorithm	code coverage, execution time
[45]	weighted sum	MC/DC coverage, error-revealing ability
[26]	genetic algorithm	function coverage, historical fault detection, execution cost
[32]	simple sorting on sum of weighted criteria	requirements' priority, code complexity, requirement, change, fault impact
[18]	multi-objective optimization min-max	requirements-to-cover, requirements-to-avoid
[17]	simple sorting on sum of weighted criteria	historical effectiveness of test case,execution history of test, previous priority of test case
[35]	redundancy based on criteria	elements in source code
[23]	particle swarm optimization	statement, branch, function
[40]	bayesian network	quality metrics, code change, test case coverage
[60]	simple ordering function based on criteria	subjective criteria based on expert and objective criteria based on historical data
[47]	simple ordering function based on criteria	customer priority, changes in requirement, implementation complexity, usability, application flow and fault impact
[44]	simple sorting on sum of weighted criteria	coverage degree of test case for test requirements, capability of test cases to reveal error
[66]	genetic search	block based coverage, test-execution cost
[54]	ant colony optimization	test case fault coverage, execution time
[38]	genetic algorithms	program element coverage
[1]	greedy, dynamic programming, generalized tabular, core	code coverage, execution time
	Rank	
[63]	genetic	execution time, code coverage (block and method)
[21]	greedy	program specifications, all-uses
[28]	greedy	branch, def-use
[30]	greedy	black-box, branches, def-use
[52]	greedy	statement, method, conditional, and usage-based
[36]	greedy	branch, def-use
[3]	greedy	usage-based, event-based
[14]	greedy	estimate of fault exposing potential and fault proneness, fault index and fault exposing potential
[19]	greedy	control flow-based and data-flow dependency
[24]	integer linear programming	branch coverage and time constraints based on request quotas (constraints)
[34]	greedy	statement coverage and historical fault detection
[73]	greedy	criteria based on code elements
[70]	agglomerative hierarchical clustering	structural coverage and fault detection information
[72]	integer linear programming	total and additional statement coverage and time-aware
[10]	bayesian network and greedy	class level coverage, block-level coverage, time constraints
	Higher order hybrid	
[61]	machine learning	user input, statement coverage, cyclomatic complexity
[64]	greedy	fault classes

This result is an artifact of our applications and the faults seeded in them. Our focus in this paper is primarily to show how the hybrids can be based on our formalizations and applied for the test case prioritization problem.

In general, we find that no one particular hybrid criterion performs best across all our subject applications. In our experiments, we find that a hybrid criterion that combines several individual criteria performs better than a single criterion, because we do not consider time costs in the evaluation metrics, and hybrid criteria will usually be more expensive than a single criterion because they consider several factors during prioritization.

5 Conclusions and Future Work

We formalized the notion of "hybrid criteria" by creating a uniform representation to precisely describe them. As a starting point, we represented three hybrids: 1) *Rank* that gives primary, secondary, and n-order precedence to different criteria, 2) *Merge* that applies multiple criteria simultaneously, and 3) *Choice* that selects between multiple criteria.

Our three demonstrations showed several things. First, that others' past work can be described using the *Merge* and *Rank* formulations. Second, our empirical study on several GUI and web applications for the problem of test case prioritization using *Merge*, *Rank*, and *Choice* formulations showed the ease at which our formalization can be used. Third, we show scenarios in which hybrid criteria are beneficial.

In future work, we anticipate that many techniques that use a single criterion will be revisited to evaluate the application of multiple criteria. Future work may also examine the relationship of multiple criteria to different techniques (i.e., test case generation, regression test selection, test suite reduction, and test case prioritization) on applications with different characteristics so that there is a better understanding of the relationship of multiple criteria in different scenarios. We anticipate that future work may identify new types of hybrid combinations

beyond *Rank*, *Merge*, and *Choice*. We may also explore combining these operators with the goal of finding an optimal hybrid approach that maximizes some user-specified criteria. We assumed that there are no dependencies between test cases. This is a common assumption in most regression testing literature. However, we can generalize by relaxing this assumption. Additional constraints will need to be added that ensure certain test cases are run before others that depend on their results. Finally, we anticipate that the framework provides a step toward helping researchers to create shared tools and artifacts that use a uniform representation.

APPENDIX

Table 5 shows the papers we studied as part of the recasting of previous work on hybrid criteria for performing regression testing tasks. The papers in the table are categorized by "Merge," "Rank," "Higher order hybrid" (note that we did not find previous work that was an application of our "Choice" formalization). For each paper referenced in Column 1, we list the keywords in Column 2 that capture the essence of the hybrid method that is presented and the criteria that are combined to create the hybrid in Column 3.

ACKNOWLEDGMENTS

The authors thank the reviewers, who provided valuable input that helped to shape this paper. They appreciate partial support for this work that came from the US National Science Foundation Award CNS-1205501 and the US National Institute of Standards and Technology Award 70NANB10H048.

REFERENCES

[1] S. Alspaugh, K.R. Walcott, M. Belanich, G.M. Kapfhammer, and M.L. Soffa, "Efficient Time-Aware Prioritization with Knapsack Solvers," *Proc. First ACM Int'l Workshop Empirical Assessment of Software Eng. Languages and Technologies: Held in Conjunction with the 22nd IEEE/ACM Int'l Conf. Automated Software Eng.*, pp. 13-18, 2007.

[2] J. Black, E. Melachrinoudis, and D. Kaeli, "Bi-Criteria Models for All-Uses Test Suite Reduction," *Proc. Int'l Conf. Software Eng.*, pp. 106-115, May 2004.

[3] R. Bryce, S. Sampath, and A. Memon, "Developing a Single Model and Test Prioritization Strategies for Event-Driven Software," *IEEE Trans. Software Eng.*, vol. 37, no. 1, pp. 48-64, Jan./Feb. 2011.

[4] R. Carlson, H. Do, and A. Denton, "A Clustering Approach to Improving Test Case Prioritization: An Industrial Case Study," *Proc. 27th IEEE Int'l Conf. Software Maintenance*, pp. 382-391, Sept. 2011.

[5] T.Y. Chen and M.F. Lau, "Dividing Strategies for the Optimization of a Test Suite," *Information Processing Letters*, vol. 60, no. 3, pp. 135-141, Mar. 1996.

[6] T.Y. Chen and M.F. Lau, "A New Heuristic for Test Suite Reduction," *Information and Software Technology*, vol. 40, nos. 5/6, pp. 347-354, Sept. 1998.

[7] A.D. Lucia, M.D Penta, R. Oliveto, and A. Panichella, "On the Role of Diversity Measures for Multi-Objective Test Case Selection," *Proc. Seventh Int'l Workshop Automation of Software Test*, pp. 145-151, June 2012.

[8] L.S. de Souza, P.B.C. de Miranda, R.B.C. Prudencio, and F.A. de Barros, "A Multi-Objective Particle Swarm Optimization for Test Case Selection Based on Functional Requirements Coverage and Execution Effort," *Proc. 23rd IEEE Int'l Conf. Tools with Artificial Intelligence*, pp. 245-252, Nov. 2011.

[9] H. Do, S. Mirarab, L. Tahvildari, and G. Rothermel, "The Effects of Time Constraints on Test Case Prioritization: A Series of Controlled Experiments," *IEEE Trans. Software Eng.*, vol. 36, no. 5, pp. 593-617, Oct./Nov. 2010.

[10] 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," *Proc. 16th ACM SIGSOFT Int'l Symp. Foundations of Software Eng.*, pp. 71-82, 2008.

[11] C. Donghua and Y. Wenjie, "The Research of Test-Suite Reduction Technique," *Proc. Int'l Conf. Consumer Electronics, Comm. and Networks*, pp. 4552-4554, Apr. 2011.

[12] S. Elbaum, D. Gable, and G. Rothermel, "Understanding and Measuring the Sources of Variation in the Prioritization of Regression Test Suites," *Proc. Int'l Software Metrics Symp.*, pp. 169-179, Apr. 2001.

[13] S. Elbaum, A. Malishevsky, and G. Rothermel, "Incorporating Varying Test Costs and Fault Severities into Test Case Prioritization," *Proc. 23rd Int'l Conf. Software Eng.*, pp. 329-338, 2001.

[14] S. Elbaum, A.G. Malishevsky, and G. Rothermel, "Prioritizing Test Cases for Regression Testing," *Proc. ACM SIGSOFT Int'l Symp. Software Testing and Analysis*, pp. 102-112, 2000.

[15] S. Elbaum, A.G. Malishevsky, and G. Rothermel, "Test Case Prioritization: A Family of Empirical Studies," *IEEE Trans. Software Eng.*, vol. 28, no. 2, pp. 159-182, Feb. 2002.

[16] S. Elbaum, G. Rothermel, S. Kanduri, and A. Malishevsky, "Selecting a Cost-Effective Test Case Prioritization Technique," *Software Quality J.*, vol. 12, no. 3, pp. 185-210, Sept. 2004.

[17] Y. Fazlalizadeh, A. Khalilian, M.A. Azgomi, and S. Parsa, "Prioritizing Test Cases for Resource Constraint Environments Using Historical Test Case Performance Data," *Proc. Second IEEE Int'l Conf. Computer Science and Information Technology*, pp. 190-195, Aug. 2009.

[18] Q. Gu, B. Tang, and D. Chen, "Optimal Regression Testing Based on Selective Coverage of Test Requirements," *Proc. Int'l Symp. Parallel and Distributed Processing with Applications*, pp. 419-426, Sept. 2010.

[19] R.A. Haraty, N. Mansour, and B. Daou, "Regression Testing of Database Applications," *Proc. ACM Symp. Applied Computing*, pp. 285-289, 2001.

[20] M. Harman, "Making the Case for MORTO: Multi Objective Regression Test Optimization," *Proc. First Int'l Workshop Regression Testing*, pp. 111-114, 2011.

[21] M.J. Harrold, R. Gupta, and M.L. Soffa, "A Methodology for Controlling the Size of a Test Suite," *ACM Trans. Software Eng. and Methodology*, vol. 2, no. 3, pp. 270-285, July 1993.

[22] M.P.E. Heimdahl and D. George, "Test-Suite Reduction for Model Based Tests: Effects on Test Quality and Implications for Testing," *Proc. Int'l Conf. Automated Software Eng.*, pp. 176-185, Sept. 2004.

[23] K.H.S. Hla, Y. Choi, and J.S. Park, "Applying Particle Swarm Optimization to Prioritizing Test Cases for Embedded Real Time Software Retesting," *Proc. Eighth IEEE Int'l Conf. Computer and Information Technology Workshops*, pp. 527-532, July 2008.

[24] S.-S. Hou, L. Zhang, T. Xie, and J.-S. Sun, "Quota-Constrained Test-Case Prioritization for Regression Testing of Service-Centric Systems," *Proc. IEEE Int'l Conf. Software Maintenance*, pp. 257-266, Oct. 2008.

[25] H.-Y. Hsu and A. Orso, "MINTS: A General Framework and Tool for Supporting Test-Suite Minimization," *Proc. 31st IEEE and ACM SIGSOFT Int'l Conf. Software Eng.*, pp. 419-429, May 2009.

[26] Y.-C. Huang, C.-Y. Huang, J.-R. Chang, and T.-Y. Chen, "Design and Analysis of Cost-Cognizant Test Case Prioritization Using Genetic Algorithm with Test History," *Proc. 34th IEEE Ann. Computer Software and Applications Conf.* pp. 413-418, July 2010.

[27] M.M. Islam, A. Marchetto, A. Susi, and G. Scanniello, "A Multi-Objective Technique to Prioritize Test Cases Based on Latent Semantic Indexing," *Proc. 16th European Conf. Software Maintenance and Reeng.*, pp. 21-30, Mar. 2012.

[28] D. Jeffrey and N. Gupta, "Test Suite Reduction with Selective Redundancy," *Proc. IEEE Int'l Conf. Software Maintenance*, pp. 549-558, 2005.

[29] D. Jeffrey and N. Gupta, "Test Case Prioritization Using Relevant Slices," *Proc. Int'l Computer Software and Applications Conf.*, pp. 411-418, Sept. 2006.

[30] D. Jeffrey and N. Gupta, "Improving Fault Detection Capability by Selectively Retaining Test Cases during Test Suite Reduction," *IEEE Trans. Software Eng.*, vol. 33, no. 2, pp. 108-123, Feb. 2007.

[31] J.A. Jones and M.J. Harrold, "Test Suite Reduction and Prioritization for Modified Condition/Decision Coverage," *IEEE Trans. Software Eng.*, vol. 29, no. 3, pp. 195-209, Mar. 2003.
[32] R. Kavitha, V.R. Kavitha, and N.S. Kumar, "Requirement Based Test Case Prioritization," *Proc. IEEE Int'l Conf. Comm. Control and Computing Technologies*, pp. 826-829, Oct. 2010.
[33] J.M. Kim and A. Porter, "A History-Based Test Prioritization Technique for Regression Testing in Resource Constrained Environments," *Proc. Int'l Conf. Software Eng.*, pp. 119-129, May 2002.
[34] S. Kim and J. Baik, "An Effective Fault Aware Test Case Prioritization by Incorporating a Fault Localization Technique," *Proc. ACM/IEEE Int'l Symp. Empirical Software Eng. and Measurement*, pp. 5:1-5:10, 2010.
[35] N. Koochakzadeh, V. Garousi, and F. Maurer, "Test Redundancy Measurement Based on Coverage Information: Evaluations and Lessons Learned," *Proc. Int'l Conf. Software Testing Verification and Validation*, pp. 220-229, Apr. 2009.
[36] J. Lin and C. Huang, "Analysis of Test Suite Reduction with Enhanced Tie-Breaking Techniques," *Information and Software Technology*, vol. 51, no. 11, pp. 679-690, 2008.
[37] C. Malz and P. Gohner, "Agent-Based Test Case Prioritization," *Proc. IEEE Fourth Int'l Conf. Software Testing, Verification and Validation Workshops*, pp. 149-152, Mar. 2011.
[38] W. Masri and M. El-Ghali, "Test Case Filtering and Prioritization Based on Coverage of Combinations of Program Elements," *Proc. Seventh Int'l Workshop Dynamic Analysis*, pp. 29-34, 2009.
[39] S. McMaster and A.M. Memon, "Call-Stack Coverage for GUI Test-Suite Reduction," *IEEE Trans. Software Eng.*, vol. 34, no. 1, pp. 99-115, Jan. 2008.
[40] S. Mirarab and L. Tahvildari, "An Empirical Study on Bayesian Network-Based Approach for Test Case Prioritization," *Proc. First Int'l Conf. Software Testing, Verification, and Validation*, pp. 278-287, Apr. 2008.
[41] S. Mirarab, S. Akhlaghi, and L. Tahvildari, "Size-Constrained Regression Test Case Selection Using Multi-Criteria Optimization," *IEEE Trans. Software Eng.*, vol. 38, no. 4, pp. 936-956, July/Aug. 2012.
[42] S. Nachiyappan, A. Vimaladevi, and C.B. SelvaLakshmi, "An Evolutionary Algorithm for Regression Test Suite Reduction," *Proc. Int'l Conf. Comm. and Computational Intelligence*, pp. 503-508, Dec. 2010.
[43] R.L. Ott and M. Longnecker, *An Introduction to Statistical Methods and Data Analysis*, fifth ed. Duxbury, 2001.
[44] L. Pan, B. Zou, J. Li, and H. Chen, "Bi-Objective Model for Test-Suite Reduction Based on Modified Condition/Decision Coverage," *Proc. 11th Pacific Rim Int'l Symp. Dependable Computing*, p. 7, Dec. 2005.
[45] J. Prabhu, N. Malmurugan, G. Gunasekaran, and R. Gowtham, "Study of ERP Test-Suite Reduction Based on Modified Condition/Decision Coverage," *Proc. Second Int'l Conf. Computer Research and Development*, pp. 373-378, May 2010.
[46] X. Qu, M.B. Cohen, and K.M. Woolf, "Combinatorial Interaction Regression Testing: A Study of Test Case Generation and Prioritization," *Proc. Int'l Conf. Software Maintenance*, pp. 255-264, Oct. 2007.
[47] K. Ramasamy and S.A. Mary, "Incorporating Varying Requirement Priorities and Costs in Test Case Prioritization for New and Regression Testing," *Proc. Int'l Conf. Computing, Comm., and Networking*, pp. 1-9, Dec. 2008.
[48] G. Rothermel, M.J. Harrold, J. von Ronne, and C. Hong, "Empirical Studies of Test Suite Reduction," *J. Software Testing, Verification, and Reliability*, vol. 12, no. 4, pp. 219-249, Dec. 2002.
[49] G. Rothermel, R.H. Untch, C. Chu, and M.J. Harrold, "Prioritizing Test Cases for Regression Testing," *IEEE Trans. Software Eng.*, vol. 27, no. 10, pp. 929-948, Oct. 2001.
[50] G. Rothermel, R.J. Untch, and C. Chu, "Prioritizing Test Cases for Regression Testing," *IEEE Trans. Software Eng.*, vol. 27, no. 10, pp. 929-948, Oct. 2001.
[51] S. Sampath and R.C. Bryce, "Improving the Effectiveness of Test Suite Reduction for User-Session-Based Testing of Web Applications," *Information and Software Technology*, vol. 54, no. 7, pp. 724-738, 2012.
[52] S. Sampath, S. Sprenkle, E. Gibson, and L. Pollock, "Integrating Customized Test Requirements with Traditional Requirements in Web Application Testing," *Proc. Workshop Testing, Analysis, and Verification of Web Services and Applications*, pp. 23-32, 2006.
[53] S. Sampath, S. Sprenkle, E. Gibson, L. Pollock, and A.S. Greenwald, "Applying Concept Analysis to User-Session-Based Testing of Web Applications," *IEEE Trans. Software Eng.*, vol. 33, no. 10, pp. 643-658, Oct. 2007.
[54] Y. Singh, A. Kaur, and B. Suri, "Test Case Prioritization Using Ant Colony Optimization," *SIGSOFT Software Eng. Notes*, vol. 35, no. 4, pp. 1-7, July 2010.
[55] *Software-Artifact Infrastructure Repository (SIR)*, http://sir.unl.edu/portal/index.php, 2012.
[56] A.M. Smith and G.M. Kapfhammer, "An Empirical Study of Incorporating Cost into Test Suite Reduction and Prioritization," *Proc. ACM Symp. Applied Computing*, pp. 461-467, 2009.
[57] S. Sprenkle, E. Gibson, S. Sampath, and L. Pollock, "Automated Replay and Failure Detection for Web Applications," *Proc. Int'l Conf. Automated Software Eng.*, pp. 253-262, Nov. 2005.
[58] S. Sprenkle, S. Sampath, E. Gibson, L. Pollock, and A. Souter, "An Empirical Comparison of Test Suite Reduction Techniques for User-Session-Based Testing of Web Applications," *Proc. Int'l Conf. Software Maintenance*, pp. 587-596, Sept. 2005.
[59] A. Srivastava and J. Thiagarajan, "Effectively Prioritizing Tests in Development Environment," *Proc. ACM SIGSOFT Int'l Symp. Software Testing and Analysis*, pp. 97-106, 2002.
[60] M.G. Stochel and R. Sztando, "Testing Optimization for Mission-Critical, Complex, Distributed Systems," *Proc. 32nd Ann. IEEE Int'l Computer Software and Applications*, pp. 847-852, Aug. 2008.
[61] P. Tonella, P. Avesani, and A. Susi, "Using the Case-Based Prioritization," *Proc. 22nd IEEE Int'l Conf. Software Maintenance*, pp. 123-133, Sept. 2006.
[62] M. Utting and B. Legeard, *Practical Model-Based Testing: A Tools Approach*. Morgan Kaufmann Publishers Inc., 2007.
[63] K.R. Walcott, M.L. Soffa, G.M. Kapfhammer, and R. Roos, "Time-Aware Test Suite Prioritization," *Proc. Int'l Symp. Software Testing and Analysis*, pp. 1-12, July 2006.
[64] Z. Wang, Z. Chen, T.-Y. Chen, and B. Xu, "Fault Class Prioritization in Boolean Expressions," *Proc. 27th Ann. ACM Symp. Applied Computing*, pp. 1191-1196, 2012.
[65] W.E. Wong, J.R. Horgan, S. London, and A.P. Mathur, "Effect of Test Set Minimization on Fault Detection Effectiveness," *Proc. Int'l Conf. Software Eng.*, pp. 41-50, 1995.
[66] X.-y. Ma, Z.-f. He, B.-k. Sheng, and C.-q. Ye, "A Genetic Algorithm for Test-Suite Reduction," *Proc. IEEE Int'l Conf. Systems, Man, and Cybernetics*, vol. 1, pp. 133-139, Oct. 2005.
[67] S. Yoo and M. Harman, "Pareto Efficient Multi-Objective Test Case Selection," *Proc. Int'l Symp. Software Testing and Analysis*, pp. 140-150, July 2007.
[68] S. Yoo and M. Harman, "Using Hybrid Algorithm for Pareto Efficient Multi-Objective Test Suite Minimisation," *J. Systems and Software*, vol. 83, no. 4, pp. 689-701, Apr. 2010.
[69] S. Yoo and M. Harman, "Regression Testing Minimisation, Selection and Prioritisation: A Survey," *J. Software Testing, Verification, and Reliability*, vol. 22, no. 2, pp. 67-120, 2012.
[70] S. Yoo, M. Harman, P. Tonella, and A. Susi, "Clustering Test Cases to Achieve Effective and Scalable Prioritisation Incorporating Expert Knowledge," *Proc. 18th Int'l Symp. Software Testing and Analysis*, pp. 201-212, 2009.
[71] L. Zhang, J. Zhou, D. Hao, L. Zhang, and H. Mei, "Prioritizing Junit Test Cases in Absence of Coverage Information," *Proc. Int'l Conf. Software Maintenance*, pp. 19-28, Sept. 2009.
[72] L. Zhang, S.-S. Hou, C. Guo, T. Xie, and H. Mei, "Time-Aware Test-Case Prioritization Using Integer Linear Programming," *Proc. 18th Int'l Symp. Software Testing and Analysis*, pp. 213-224, 2009.
[73] X. Zhang, Q. Gu, X. Chen, J. Qi, and D. Chen, "A Study of Relative Redundancy in Test-Suite Reduction while Retaining or Improving Fault-Localization Effectiveness," *Proc. ACM Symp. Applied Computing*, pp. 2229-2236, 2010.
[74] H. Zhu, P.A.V. Hall, and J.H.R. May, "Software Unit Test Coverage and Adequacy," *ACM Computing Surveys*, vol. 29, no. 4, pp. 366-427, Dec. 1997.

Sreedevi Sampath received the BE degree in computer science and engineering from Osmania University in 2000 and the MS and PhD degrees in computer and information sciences from the University of Delaware in 2002 and 2006, respectively. She is an assistant professor in the Department of Information Systems, University of Maryland, Baltimore County. Her research interests are in the areas of regression testing, test case generation, web applications, and software maintenance. She has served on the program committees of conferences, such as the International Conference on Software Testing Verification and Validation, and the International Conference on Empirical Software Engineering and Measurement. She is a member of the IEEE.

Renée Bryce received the BS and MS degrees from the Rensselaer Polytechnic Institute and the PhD degree from Arizona State University. She is an associate professor at the University of North Texas. Her research interests include software testing, particularly combinatorial testing, test suite prioritization, and usability testing. She is an area editor for software for *Computer*. She has served on the program committee of the International Conference on Software Testing Verification and Validation and the International Workshop on Testing Techniques & Experimentation Benchmarks for Event-Driven Software. She is a member of the IEEE.

Atif M. Memon is an associate professor in the Department of Computer Science, University of Maryland. His research interests include program testing, software engineering, artificial intelligence, plan generation, reverse engineering, and program structures. He is the inventor of the GUITAR system (http://guitar.sourceforge.net/) for automated model-based GUI testing. He is the founder of the International Workshop on Testing Techniques & Experimentation Benchmarks for Event-Driven Software. He serves on various editorial boards, including that of the *Journal of Software Testing, Verification, and Reliability*. He has served on numerous US National Science Foundation panels and program committees, including the International Conference on Software Engineering, International Symposium on the Foundations of Software Engineering, International Conference on Software Testing Verification and Validation, the Web Engineering Track of the International World Wide Web Conference, the Working Conference on Reverse Engineering, the International Conference on Automated Software Engineering, and the International Conference on Software Maintenance. He is currently serving on a National Academy of Sciences panel as an expert in the area of computer science and information technology, for the Pakistan-US Science and Technology Cooperative Program, sponsored by US Agency for International Development. He is a senior member of the IEEE.