

# A Study in Prioritization for Higher Strength Combinatorial Testing

Xiao Qu

Industrial Software System Department  
ABB Corporate Research  
Raleigh, NC  
xiao.qu@us.abb.com

Myra B. Cohen

Department of Computer Science and Engineering  
University of Nebraska - Lincoln  
Lincoln, NE  
myra@cse.unl.edu

**Abstract**—Recent studies have shown that combinatorial interaction testing (CIT) is an effective fault detection technique and that early fault detection can be improved by ordering test suites by *interaction based prioritization* approaches. Despite research that has shown that higher strength CIT improves fault detection, there have been fewer studies that aim to understand the impact of prioritization based on higher strength criteria. In this paper, we aim to understand how interaction based prioritization techniques perform, in terms of early fault detection when we prioritize based on 3-way interactions. We generalize prior work on prioritizing using 2-way interactions to  $t$ -way prioritization, and empirically evaluate this on three open source subjects, across multiple versions of each. We examine techniques that prioritize both existing CIT suites as well as generate new ones in prioritized order. We find that early fault detection can be improved when prioritizing 3-way CIT test suites by interactions that cover more code, and to a lesser degree when generating tests in prioritized order. Our techniques that work only from the specification, appear to work best with 2-way generation.

## I. INTRODUCTION

Regression testing is an expensive part of the software process. As systems evolve, before new versions are released, software must be re-tested to ensure quality. One concern in regression testing is the effectiveness of test suites in finding new faults in successive program versions. A second issue is the efficiency of running the test suites given limited resources and time. Small test suites that retain high fault detection ability are desirable. A focus of regression testing research has been the reduction of test suite size between versions, which can be accomplished through test suite selection [1]. A further improvement, once tests are selected, is to order or *prioritize* [2] test cases to increase the likelihood of faults being discovered early in the test process. Detecting faults early, means that work to repair faults can begin sooner, and if resources are exhausted before all tests complete, the consequences are less severe.

One specification based technique for generating test suites is combinatorial interaction testing [3]–[6] or *CIT*. In CIT the program is divided into partitions or *factors*, and a test suite is generated from this: factors are tested together so that all  $t$ -way combinations appear at least once ( $t$  is called *testing strength*). This technique has been shown to produce

small test suites, with high code coverage, that exhibit good fault detection ability [4], [6], [7].

Although CIT has shown to be an effective test generation technique, it has mostly been examined in the context of single version software systems. In recent work we examined the ability to effectively utilize CIT in regression testing, from both the test case [8] and configuration level [9] of abstraction. Our results have shown that CIT is effective when used in regression testing for multiple versions of a program and that interaction based prioritization improves early fault detection.

However, our results have also shown that 2-way CIT test suites fail to detect all faults that can be detected by CIT test suites with higher testing strengths. We saw this in seven of ten versions [8]. This leaves us with several questions. We would like to understand *how* to prioritize CIT test suites with strength greater than 2, and we would like to understand if higher strength oriented prioritization can improve the effectiveness of these test suites. Finally, we would like to understand if different techniques for prioritization impact its effectiveness at higher strengths.

In this paper we address each of these issues. We have conducted an empirical study on three software subjects, each with multiple successive versions. We apply both (pure) prioritization and regeneration<sup>1</sup> [8], for both 2-way and 3-way CIT and compare their effectiveness. We examine several different ways to control the prioritization. We use methods that weight interactions based on branch coverage from prior releases, as well as one that is specification based (and does not require prior versions). Our results show that the 3-way CIT test suites prioritized by weighting interactions based on branch coverage can detect faults more efficiently than their respective 2-way prioritized test suites. Regeneration, while eventually more effective at 3-way, has a slower growth rate at the start, suggesting that we should use it only if time allows. Finally, when code coverage information is unavailable, our 2-way specification based regeneration approach may be the best choice.

The rest of this paper is organized as follows. Section

<sup>1</sup>Pure prioritization and regeneration are different interaction based prioritization approaches. In the rest of the paper, we just use prioritization and regeneration for short.

II presents background and related work on regression testing, combinatorial interaction testing, the interaction based regeneration approach, and the metric we developed to evaluate the prioritization. Section III discusses work on prioritization algorithms for CIT of different testing strengths. Section IV introduces our empirical study. Section V presents our results, and Section VI concludes and presents future work.

## II. BACKGROUND AND RELATED WORK

In this section we provide some background on regression testing, combinatorial interaction testing, interaction based prioritization, and the metric used for evaluating the effectiveness of CIT-based prioritization techniques. We end with some other related work on interaction based prioritization.

### A. Regression Testing

Regression testing is performed each time a system is modified. Given an an initial version of a program  $P$ , we need to test the new version  $P'$  to ensure that new functionality works correctly and that old features are not broken. Since new tests are often generated to cover new functionality, regression test suites can grow prohibitively large. Two directions of research aim to solve this problem. In test case *selection*, given an initial version of a program  $P$  and a set of test cases,  $T$ , select a subset of tests from  $T$ ,  $T'$  to test a new version of program  $P$ ,  $P'$  [2]. Prioritization techniques [2], [10]–[12] (the focus of this paper), complement the selection technique. Test cases are ordered to improve the likelihood that faults will be detected early in the testing process. Techniques for prioritization include statement coverage, function coverage and fault finding exposure, or interaction coverage [2], [8], [10]–[13].

### B. Combinatorial Interaction Testing

The Test Specification Language (TSL) [14] is a specification based method to define the combinations of program parameters that should be tested together. TSL partitions the system inputs into parameters and environments which make up the categories. For each of the *category*, a set of *choices* is defined based on equivalence classes of the input domain.

An exhaustive test suite generated by combining each of the possible choices of categories is too large in practice. One way to subset the test cases is combinatorial interaction testing (CIT) [4]. In CIT the categories are called *factors* and each factor has a set of *values* (choices in TSL). A CIT test suite samples the input space so that it includes all  $t$ -way combinations of values between factors, where  $t$  is called the strength of testing. When  $t=2$ , we call this pair-wise testing.

CIT samples are defined by mathematical objects called covering arrays. A *covering array*,  $CA(N; t, k, v)$ , is an  $N \times k$  array on  $v$  symbols with the property that every  $N \times t$  sub-array contains all ordered subsets from  $v$  symbols of size  $t$  at least once [5]. Quite often in software testing the number

of values for each factor is not the same. Therefore, we use the following expanded definition (often called a mixed level covering array) that uses a vector of  $vs$  for the factors.

A mixed level covering array,  $CA(N; t, k, (v_1 v_2 \dots v_k))$ , is an  $N \times k$  array on  $v$  symbols, where  $v = \sum_{i=1}^k v_i$ , where each column  $i$  ( $1 \leq i \leq k$ ) contains only elements from a set  $S_i$  of size  $v_i$  and the rows of each  $N \times t$  sub-array cover all  $t$ -tuples of values from the  $t$  columns at least once. We use a shorthand notation to describe these arrays with superscripts to indicate the number of factors with a particular number of values. For example, a covering array with 5 factors, 3 of which are binary and 2 of which have four values can be written as  $CA(N; 2, 3^2 2^4)$  (we remove the  $k$  since it is implicit). Covering arrays (i.e., CIT suites) have been shown to be effective test suites in a variety of studies [3], [4], [6].

### C. Interaction based Prioritization

Bryce and Colbourn [13] describe an algorithm for *re-generating* prioritized test suites; each time a new test suite is generated from scratch. They begin by defining a set of weights for each value of each factor. Based on the defined weights, their approach generates test suites that are a special kind of a covering array, called a *biased covering array*. The pseudo-code for their algorithm is presented in Algorithm 1 in Section III-A.

An alternative way to perform prioritization is to re-order an existing CIT test suites (*pure prioritization*). One advantage of this approach is that we can re-use concrete test cases and artifacts from prior versions. We have seen in our prior work that the regenerated test suites often find faults sooner, however the overall test suite tends to be larger requiring more resources to obtain full  $t$ -way coverage [8], [9]. We believe there is a trade-off, therefore, in which of these methods to use.

### D. Prioritization Evaluation Metric: NAPFD

A metric that is commonly used for prioritization is the *Average Percentage of Faults Detected* or APFD [2], [10]. APFD measures the area under the curve when the percent of faults found is plotted on the  $y$ -axis and the percent of the test cases run on the  $x$ -axis. We have modified the APFD metric slightly to handle test suites of different size/fault finding ability [8]. We call this metric NAPFD (for normalized APFD) :

$$NAPFD = p - \frac{TF_1 + TF_2 + \dots + TF_m}{m \times n} + \frac{p}{2n}$$

where  $p$  = the number of faults detected by the prioritized test suite divided by the number of faults detected in the full test suite ( $y$ -axis). If a fault,  $i$ , is never detected we set  $TF_i = 0$ . In the example fault matrix shown in Table I, there are 5 tests and 8 faults. The ordering of the test cases is  $T_3, T_5, T_2, T_4, T_1$ . The first test finds 3 faults. After

3 test cases, 5 faults have been found. Suppose in our example, only three test cases (i.e.,  $T_3, T_5, T_2$ ) are executed, then only five faults are detected. Hence,  $p = 0.625$ , and  $TF_1 = TF_3 = TF_8 = 0$ . As a result, the NAPFD is 0.44.

	$T_1$	$T_2$	$T_3$	$T_4$	$T_5$
$F_1$				x	
$F_2$		x	x		
$F_3$				x	
$F_4$			x		x
$F_5$					x
$F_6$					x
$F_7$	x		x		
$F_8$	x				

Table I  
FAULT MATRIX (TEST ORDER:  $T_3, T_5, T_2, T_4, T_1$ )

### E. Other Related Work

In other related work, Sampath et al. and Manchester also describe a  $t$ -way prioritization technique and tool, CPUT [15], [16], which prioritizes given test cases by their coverage of (additional)  $t$ -way parameter-value interactions. This has been applied to user-session-based test suite. This differs from our work in that the prioritization is not linked to any heuristic methods that weight one parameter-value over another. It also does not aim to achieve complete CIT coverage. Recently, Chen et al. [17] propose a new metric, tuple density, for CIT suites. It considers the coverage of tuples with higher dimensions and is used to drive test selection and optimization. This metric may be used for prioritization but their work has not addressed it.

## III. PRIORITIZATION TECHNIQUES

In this section, we present our generalization of 2-way prioritization and show how this is achieved on the specific prioritization algorithms that are used in our study. We first present our regeneration approach, followed by the pure prioritization method. We then provide examples of how we associate importance with the various values of the factors and then use these to determine *interaction weights*. We provide a running example, based on the original work of Bryce and Colbourn [13] for consistency of explanation.

### A. $t$ -wise Regeneration

Algorithm 1 shows our regeneration algorithm, which is modified slightly from [13]. In the original Algorithm the interaction weight is fixed at the pair-wise level. In the generalized algorithm (Algorithm 1), testing strength  $t$  can be varied. This can be seen on lines 1, 3, and 9. It considers  $t$ -way interaction weights, while pair-wise interaction weight is just one of its instances. The details of calculating interaction weights for  $t$ -way interactions will be introduced in Section III-C.

For each factor the weight of combining it with each other  $t-1$  factors is computed as a total interaction weight (line

```

1: RemainingCombinations = AllCombinationst
2: while Remainingcombinations  $\neq$   $\emptyset$  do
3:   Compute  $t$ -way interaction weights for factors
4:   Order factors
5:   Initialize a new test with all factors unfixed
6:   for  $i = 1$  to NumOfFactors do
7:     Select HighestWeightUnfixedFactor ( $f$ )
8:     for  $j = 1$  to NumOfValuesf do
9:       Compute  $t$ -way interaction weights for values of  $f$ 
10:      Select value with highest weight to fix  $f$ 
11:    Add test case to test suite

```

**Algorithm 1:** General  $t$ -wise Regeneration Algorithm Modified from [13]

3). The factors are sorted in decreasing order of interaction weight and then each factor is filled/fixed with value for a new test in order. To fix a factor, the individual interaction weights for each of the factor's values is computed (line 9, see [13] for more details). Then it selects the value of the factor that has the greatest value interaction weight (line 10). After all factors have been fixed, a single test has been added (line 11), and the interaction weights for factors are recomputed and the process starts again (line 3). The algorithm is complete when all  $t$ -way combinations have been covered.

### B. $t$ -wise Prioritization

We can also use interaction weights in a different way from the regeneration algorithm. Rather than regenerate a new test suite each time, we simply use the interaction weights directly; we sort the test cases in decreasing order of their *test case level* interaction weights. We show this as Algorithm 2.

```

1: RemainingTCs = Alltestcasesinoriginaltestsuite
2: while RemainingTCs  $\neq$   $\emptyset$  do
3:   Compute  $t$ -way interaction weights for test cases
4:   Order test cases
5:   Select test case with the highest interaction weight
6:   Add test case to result test suite

```

**Algorithm 2:** General  $t$ -wise Prioritization Algorithm

### C. Calculation of Interaction Weights

To utilize the algorithms presented we used interaction weights calculated at three different levels: test case level (Algorithm 2), factor level and value level (Algorithm 1). In [13], they show how to calculate pair-wise *interaction weights* for factors and values, we extend the computation from "pair-wise" to " $t$ -way". We also add a third level, that of the test case. We reuse the same example in [13] to illustrate the procedure, as shown in Table II: three factors are considered, the first factor ( $f_0$ ) has four values, the second ( $f_1$ ) has three values, and the third ( $f_2$ ) has two values. Each value is labeled in Table II with a unique ID, and a weight for each is given in parentheses.

Their work does not provide a technique to find the weights in this table, yet the weighting method may play an important role in how well the algorithm performs. Before we can prioritize in practice, we need to determine these from somewhere. We call these base-value weights. They are essential for associating elements of our software with the values that should be tested first. We present two approaches we have used in prior work (and re-use here) to find these values [8].

1) *Base-value Weights:* When we have existing test artifacts from prior versions of a system, we can use code-based weightings. For code-based weightings our aim is to associate higher weights with those values that cover larger amounts of code. In [8] we tried three different code coverage based weighting schemes, from which the third one gave us the most consistent results. We use that for this study and describe it briefly next.

We begin by ordering our test suite (from the prior version of software) by greedily increasing additional branch coverage. We begin with the test case that provides the greatest stand-alone coverage and then add to this to increase our coverage as much as possible at each stage. This is commonly known as *additional branch coverage prioritization* [2]. Based on this order, we select a small set of test cases (*important test cases*) that contribute to a large percentage of the coverage (in our study, we consider test cases that provide more than 90% of the total coverage obtained in the full test suite). Let suppose the size of the selected set is  $m$ .

Because at each stage of the greedy algorithm, more than one test case may be selected, we strengthen the set of important test cases by identifying all other test cases with equivalent additional coverage at each step and add these to our set. We now have  $m$  distinct groups ordered by importance, each with a set of test cases. We then count up the occurrences of each value within the set of test cases for each group and divide by the number of test cases in that group. We then normalize the weights of values by the contribution of its groups importance (we heuristically assigned a higher importance to the first group, and reduced this slightly for each successive group). Finally, we sum these numbers by value after normalization.

When code coverage does not exist, we use a TSL based approach. We have assigned higher weights to the values of a feature (such as turning a feature on rather than turning it off) that we believe should execute more code (see [8] for details of our TSL scheme).

Once we have the base weights, we use them to calculate the  $t$ -way interaction weights in Algorithms 1 and 2.

2) *t-way Factor Level Interaction Weights:* We use three steps to compute a  $t$ -way interaction weights of a given factor,  $f$ . First, for each value of the factor, find out its  $t$ -way combinations with all the values of other factors. Second, for each of the  $t$ -way combination identified in the previous

Factor	$v_0$	$v_1$	$v_2$	$v_3$
$f_0$	0 (.2)	1 (.1)	2 (.1)	3 (.1)
$f_1$	4 (.2)	5 (.3)	6 (.3)	
$f_2$	7 (.1)	8 (.9)		

Table II  
INPUT OF THREE FACTORS AND THEIR LEVELS AND WEIGHTS [13]

step, multiply the weight of each value that composes the combination. Finally, add up the products calculated in the previous step. For example, for  $f_0$  with  $t = 2$ , there are 20 combinations (pairs) in total, including (0,4), (0,5), (0,6), (0,7), (0,8), (1,4), ..., (3,7), and (3,8). The weight of each pair is  $0.2 \times 0.2$ ,  $0.2 \times 0.3$ , ..., and  $0.1 \times 0.9$ , as a result, the interaction weight of  $f_0$  is  $0.04 + 0.06 + \dots + 0.09 = 0.9$ . Following the same procedure, the interaction weights of  $f_1$  and  $f_2$  are 1.2 and 1.3, respectively. The maximum value among them is denoted as  $w_{max}$ , which is used in the calculation of value level interaction weights in the next section. In this case,  $w_{max} = 1.3$ .

If using  $t = 3$ , there will be 24 3-way combinations in total, including (0,4,7), (0,4,8), (0,5,7), (0,5,8), (0,6,7), (0,6,8), ..., (3,6,7), and (3,6,8). In this case, the interaction weight of  $f_0$  is equal to  $0.2 \times 0.2 \times 0.2 + 0.2 \times 0.2 \times 0.9 + \dots + 0.1 \times 0.3 \times 0.9 = 0.4$ . Similarly, the interaction weights of  $f_1$  and  $f_2$  are 0.4 and 0.4, respectively, and  $w_{max} = 0.4$ .

3) *t-way Value Level Interaction Weights:* The process of calculating value level interaction weights is similar to that of factor level interaction weights, for a given factor,  $f$ , and its value  $v$ . First, find out all of  $v$ 's  $t$ -way combinations with all the values of other factors. Second, for each of the  $t$ -way combination identified in the previous step, multiply the weight of each value that composes the combination, and divide it by the maximum value of the factor level interaction weight (i.e.,  $w_{max}$ ). Finally, add up the products calculated in the previous step. For example, for  $f_2$ ,  $v_1$  (i.e., value 8) with  $t = 2$ , there are seven pairs, which are (0,8), (1,8), (2,8), ..., (5,8), and (6,8). The weight of each pair is  $0.2 \times 0.9 \div 1.3$ ,  $0.1 \times 0.9 \div 1.3$ , ...,  $0.3 \times 0.9 \div 1.3$ , and  $0.3 \times 0.9 \div 1.3$ . As a result, the interaction weight of value 8 is  $0.18 \div 1.3 + 0.09 \div 1.3 + \dots + 0.27 \div 1.3 = 0.1$ .

4) *t-way Test Level Interaction Weights:* Interaction weights at the test case level is unique to pure prioritization. First, for each test case, find out all  $t$ -way combinations. Second, for each combination, multiply the associated weight of each value of this combination. Finally, add up all products.

Calculations at these three levels have one thing in common, which is very important. Each time a test case is selected or re-generated, the interaction weight associated with each  $t$ -way combination covered in that test case will be considered zero it doesn't need to be covered in later iterations. Accordingly, the factor level and value level interaction weights will change in each iteration as the process continues.

#### IV. EMPIRICAL STUDY

In this section, we present an empirical study to evaluate the effectiveness of higher strength interaction based prioritization. We have designed experiments to answer the following three research questions:

**RQ1:** Does prioritization with a higher testing strength provide more benefit, in terms of early fault detection?

**RQ1.1:** Does the selection of different algorithms/techniques (i.e., regeneration or prioritization) impact the results?

**RQ1.2:** Does the selection of different weighting methods (i.e., coverage based or TSL based) affect the results?

The rest of this section describes our objects of analysis, our metrics, and our methodology.

##### A. Objects of Analysis

We have used three C subjects, each with multiple successive versions. The subjects were obtained from the software infrastructure repository (SIR) [18]. The first subject, `flex` is a lexical analyzer. The second subject, `make` is used to compile programs. The third subject, `grep` is a text search utility. Table III shows the uncommented lines of code for each version of the program, the number of functions and the number of changed or added functions between versions. We used the `SLOCCOUNT` tool to count the uncommented lines of code and the `adiff` utility from the SIR to determine changed methods. These were manually verified for the purposes of seeding new faults. Each subject came with a TSL test suite and between 5-20 hand seeded faults. Hand seeding allows us to turn faults on and off during experimentation. The number of seeded faults are also shown in Table III.

To increase our ability to reason about the final results, we seeded an additional 30 faults into each subject, using a C mutation test case generator written by Andrews et al. [19]. Their research indicates that mutation faults produce similar results in empirical studies as hand seeded faults. We first generated all mutants for each program. To simulate a regression environment, we then identified the changed and added functions between consecutive versions of the programs and selected only mutants contained in those areas of the code. Finally we randomly selected 30 that successfully compiled.

We reused the TSL files in our previous study for `flex` and `make` [8], and created new reduced TSL files for `grep`, which were unconstrained. We have retained the most widely used features in each subject (based on the man pages) and ran some experiments using the original set of hand-seeded faults. We removed constraints (such as error and single) from the TSL; our objective was to obtain exhaustive suites that retain the fault detection ability that is close to the original test suite provided by the SIR. We wanted the exhaustive test suites so that we have the ground truth about the detectable faults in our study.

Subject	uLoC	Function Count	# Changed Functions	Seeded Faults
<code>flex</code>				
V0	7,972	138	NA	NA
V1	8,426	147	40	49
V2	9,932	162	104	50
V3	9,965	162	24	47
V4	10,055	162	16	46
V5	10,060	162	13	39
<code>make</code>				
V0	12,612	188	NA	NA
V1	13,484	190	80	38
V2	14,014	206	88	36
V3	14,596	239	158	35
V4	17,155	270	145	35
<code>grep</code>				
V0	7,987	132	NA	NA
V1	9,421	138	51	48
V2	9,998	144	24	38
V3	10,087	146	29	48
V4	10,128	146	18	42
V5	10,128	146	2	31

Table III  
TEST SUBJECTS STUDIED

##### B. Independent and Dependent Variables

Our independent variables are the various test suites that are prioritized by different techniques.

First, we generated 50 2-way and 3-way CIT test suites for each subject, using simulated annealing [5]. The sizes of these arrays are shown in table IV. We then use each individual CIT test suite for program  $P$ , and apply the different techniques introduced in Section III to either regenerate or prioritize for version  $P+1$ .

For the code coverage based weighting method, we call these **p-t=n**, **r-t=n**, where **p** stands for prioritization and **r** stands for regeneration. The value of **n** is the *testing strength* of the CIT test suite that was used, that is, the strength of the interaction we are considering in prioritization. Using the same notations, for the TSL based weighting method, we call these test suites **r-tsl-t=n** and **p-tsl-t=n**.

CIT Specification	Size $t=2$	Size $t=3$
<code>flex</code> $CA(N; t, 2^4 3^1 16^1 6^1)$	96	288
<code>make</code> $CA(N; t, 3^1 2^2 5^1 3^2 2^1 4^1)$	20	60
<code>grep</code> $CA(N; t, 4^1 3^1 2^1 3^1 2^1 12^1 4^1)$	48	192

Table IV  
SIZE OF CIT TEST SUITES

In total we have two different prioritization techniques, two different weighting methods, and two different testing strengths, so there are four prioritization and four regeneration techniques as follows:

- **p-t=2:** pair-wise coverage based prioritization

- **r-t=2**: pair-wise coverage based regeneration
- **p-t=3**: 3-way coverage based prioritization
- **r-t=3**: 3-way coverage based regeneration
- **p-tsl-t=2**: pair-wise TSL based prioritization
- **r-tsl-t=2**: pair-wise TSL based regeneration
- **p-tsl-t=3**: 3-way TSL based prioritization
- **r-tsl-t=3**: 3-way TSL based regeneration

We use NAPFD (introduced in II-D) as our evaluation metric, i.e., the dependent variable. To make a fair comparison between 2-way and 3-way CIT test suites, we assume that for each subject, the testing resource is available for running a test suite of the 2-way CIT size. For example, in `flex`, we only measure the fault detection of the first 96 test cases in each test suite.

### C. Study Methodology

We first run all tests on each subject without any faults as an oracle and then turn on each fault individually. We collect branch coverage on the fault free version using the Aristotle coverage tool [20]. We only include faults in our results that occur between 0 and 50% of the time in the exhaustive test suite. Our rationale is that faults occurring more than 50 percent of the time will be very easy to find and would be eliminated during unit testing.

For data using CIT test suites, we take the average of 50 arrays to prevent biases due to chance. For all of our prioritization experiments we use program  $P$  to prioritize program  $P + 1$ . For each of our subjects we have a base version, V0, with no faults. We use this only to generate the prioritization for V1.

### D. Threats to Validity

Empirical experiments suffer from threats to validity. We have made attempts to reduce these, however, we outline the major threats here. With respect to external validity (or the threat of generalizing to other subjects) we acknowledge that we have only examined three software subjects, all of which were written in the C language. We have tried to select three different subjects, of different sizes and for each have used multiple versions of the program. But results obtained from other subjects may not match these. With respect to internal validity (or the threat that our experiments themselves suffer from mistakes) we have tried to manually cross-validate our analysis programs on small examples and have manually validated random selections from the real results. We have made every effort to ensure that these are correct. As far as construct validity, (or the threat that we may not have fairly conducted these studies) we acknowledge that there may be other metrics which are more pertinent to this study. We also note that we may have developed different unconstrained TSL definitions.

## V. RESULTS

In this section, we present the results regarding each research question, and provide a discussion on the results.

### A. 2-way vs. 3-way Prioritization

1) *Cumulative Fault Coverage*: To address RQ1, which asks if 3-way prioritized or regenerated CIT test suites exhibit earlier fault detection than 2-way, we first look at the cumulative fault coverage for test suites produced by each prioritization technique, for each version of each subject program. As a baseline, we show Table V with the total average fault detection in each test suite <sup>2</sup>.

Except for the TSL based regeneration, i.e., **r-tsl-t=2** and **r-tsl-t=3**, which resulted in only one single CIT test suite, all other techniques produce 50 instances of CIT test suites; we show their average results. Except for `flex`, all of the 3-way test suites have higher average fault detection.

Subject	Max	Min	Avg
<code>flex</code>			
V4 ( $t = 2$ )	1.00	1.00	1.00
V4 ( $t = 3$ )	1.00	1.00	1.00
<code>make</code>			
V1 ( $t = 2$ )	1.00	0.60	0.76
V1 ( $t = 3$ )	1.00	0.80	0.85
<code>grep</code>			
V2 ( $t = 2$ )	1.00	0.60	0.68
V2 ( $t = 3$ )	1.00	0.60	0.90

Table V  
PERCENTAGE OF FAULTS DETECTED BY CIT TEST SUITES WITH DIFFERENT STRENGTHS

Figure 1 shows the cumulative fault coverage of `flex` (version 4) for different testing strengths, grouped by various techniques. As described in Section IV-B, for comparison, we use a test budget equal to the smallest 2-way test suite, so we only examine the first 96 test cases (the size of the 2-way CIT suite) in `flex`.

If we examine each sub-figure in Figure 1, we find that **p-t=3** exhibits constantly earlier fault detection than **p-t=2**. For other techniques, the results are more complex. Specifically, **r-t=2** detects less faults than **r-t=3** in the first couple of test cases, but its fault detection grows faster than **r-t=3**. After test case 25, both reach their maximum fault coverage. Similarly, **p-tsl-t=3** detects more faults in the starting area than **p-tsl-t=2**, but the latter grows faster in the later stage. Please note that in this figure, **p-tsl-t=3** has a long tail and does not reach a 100% fault coverage until test case 96, but the actual 3-way CIT test suite has detected all faults (shown in Table V). Finally, technique **r-tsl-t=2** detects faults faster than **r-tsl-t=3**, but after about the 25th test case, both detect all faults.

If we compare between the best techniques from each sub-figure, **r-tsl-t=2** is the best. However, as described before, the **r-tsl-t=2** generates only one single test suite, while other lines show the average results of 50 test suites. In

<sup>2</sup>Due to space limitation, we show one example version for each subject, which represents the results of other versions.

other words, although **r-tsl-t=2** performs better than others' average results, if a test suite is randomly selected from the 50 test suites produced by other techniques, **r-tsl-t=2** may lose its winning status. This is confirmed by the NAPFD measurement discussed later.

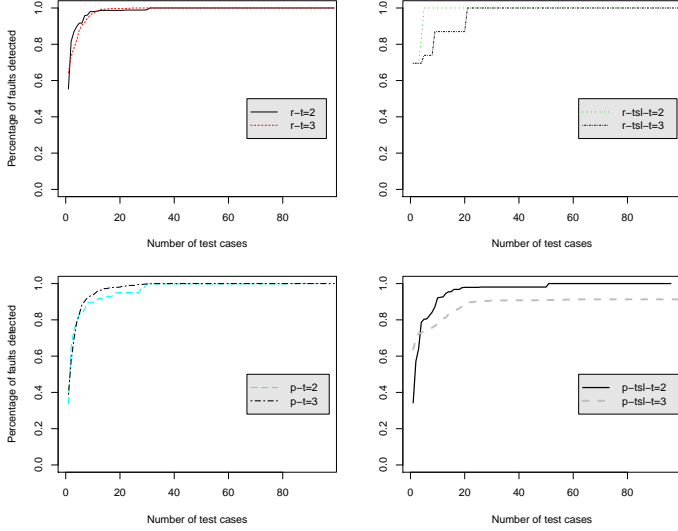


Figure 1. Cumulative Fault Coverage: flex, V4

In make (Figure 2), the result is similar to flex, except for the fact that **p-tsl-t=3** outperforms **p-tsl-t=2** not only in the first few test cases, but lasting almost to the cutting point (cut at the size of its 2-way CIT suite, which is 20).

In grep (Figure 3), all 3-way techniques exhibit benefits over pair-wise techniques, particularly after 30 test cases (the cutting point is 48 test cases). One interesting note is that **r-tsl-t=3** reaches full fault coverage while the average of other 3-way techniques do not. This may due to certain higher level (higher than 3-way) interaction related faults. As shown in Table V, the full 3-way CIT test suites can detect 90% faults on average, while the minimum fault detection is only 60%.

2) NAPFD: We examine the data next using box plots for the NAPFD (Figures 4, 5 and 6). In general, the results of NAPFD are consistent with the cumulative fault coverage information, that the 3-way performs better than pair-wise, for code coverage based prioritization.

In flex, as described in Section V-A1, technique **r-tsl-t=2** shows the best result among all eight techniques in Figure 1. But if we examine the NAPFD values in this version, many test suites produced by **r-t=2**, **r-t=3**, and **p-t=3** can achieve the same value. Moreover, if we examine other versions, we can see that NAPFD value of **r-tsl-t=2** is usually lower than the highest NAPFD of the other techniques. This confirms that **r-tsl-t=2** is only better than the average performance of other techniques, that is, it is not always the single champion.

In Figure 1, it is hard to tell which is better between **r-t=2**

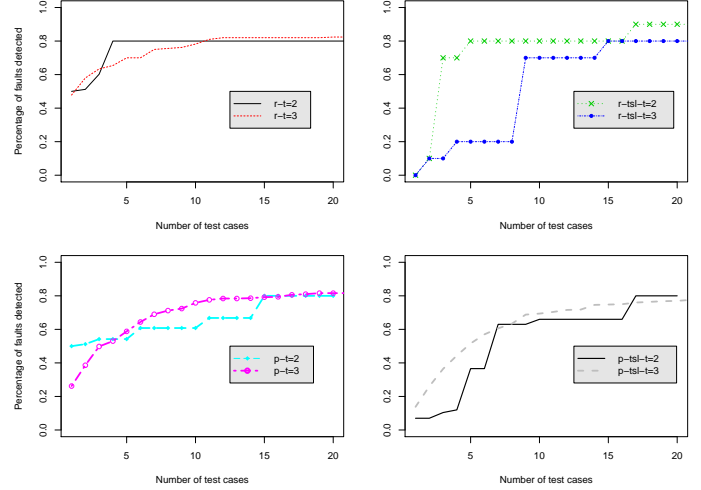


Figure 2. Cumulative Fault Coverage: make, V1

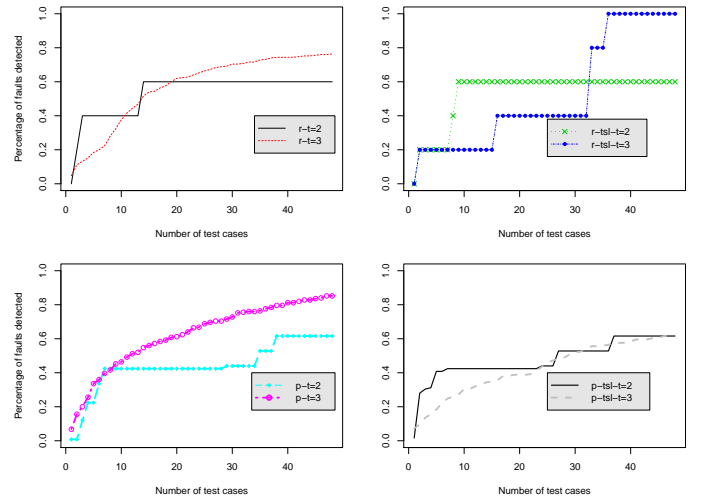


Figure 3. Cumulative Fault Coverage: grep, V2

and **r-t=3** since neither exhibits a constant advantage over the other. The NAPFD values clarify this by showing that **r-t=2** achieve similar early fault detection to **r-t=3**. Finally, given the TSL based techniques, the pair-wise prioritization or regeneration usually performs better than corresponding 3-way ones.

In make, our results are similar to flex, despite the fact that **p-tsl-t=3** outperforms **p-tsl-t=2**. This result is consistent with what we observe in Figure 2. In grep our results are more close to the results of flex and are consistent with what we observe in Figure 3.

### B. RQ1.1 Impact of prioritization techniques

If we only examine the regeneration technique (Algorithm 1), we see that the three subject programs share the same results in most versions, that **r-tsl-t=3** is not better than **r-tsl-t=2**, and **r-t=3** is not better than **r-t=2** either. Hence we conclude that, for regeneration techniques, 3-way techniques

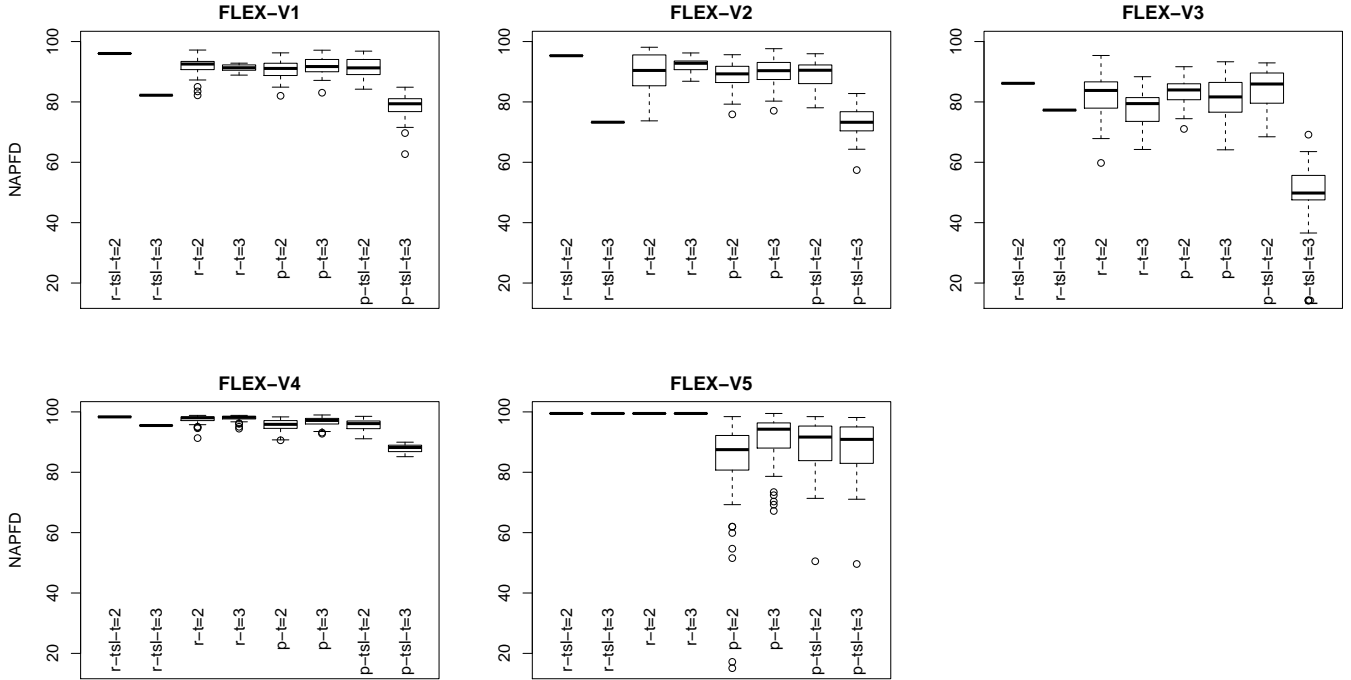


Figure 4. NAPFD for flex of different prioritization strength

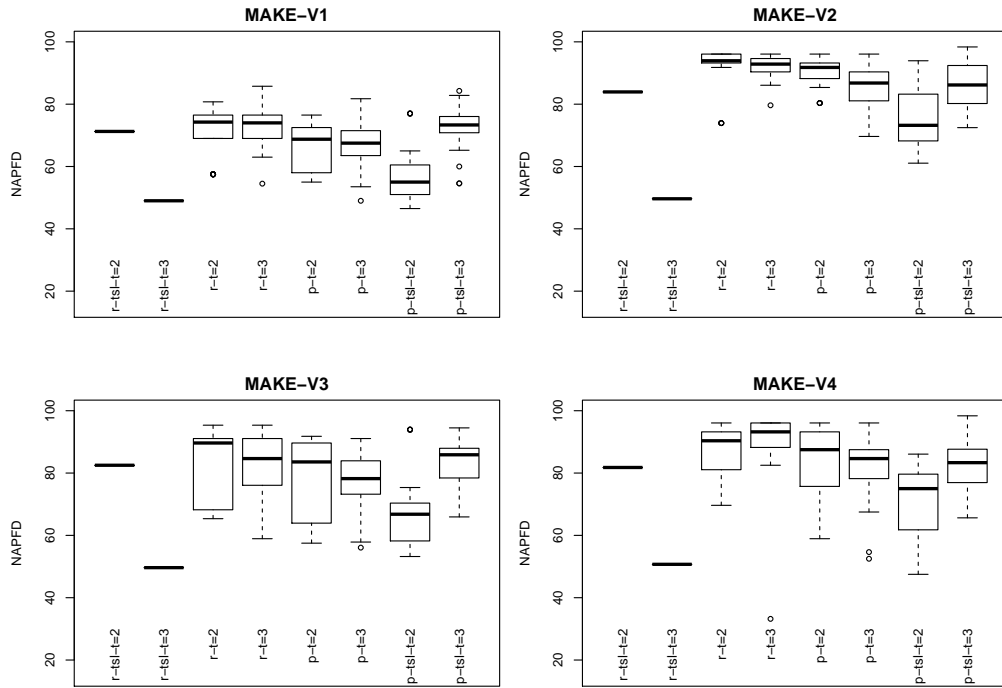


Figure 5. NAPFD for make of different prioritization strength

do not perform better than pair-wise ones, it may due to bias placed on certain 2-way interactions. More analysis will be discussed in Section V-D.

As for the prioritization techniques (Algorithm 2), in all three programs, **p-t=3** outperforms **p-t=2**, but the results of

**p-tsl-t=2** and **p-tsl-t=3** are mixed: in flex and grep, **p-tsl-t=2** is better, while in make, **p-tsl-t=3** is better.

### C. RQ1.2 Impact of weighting methods

If we examine the branch coverage based weighting, we see that the three subject programs share the same results,



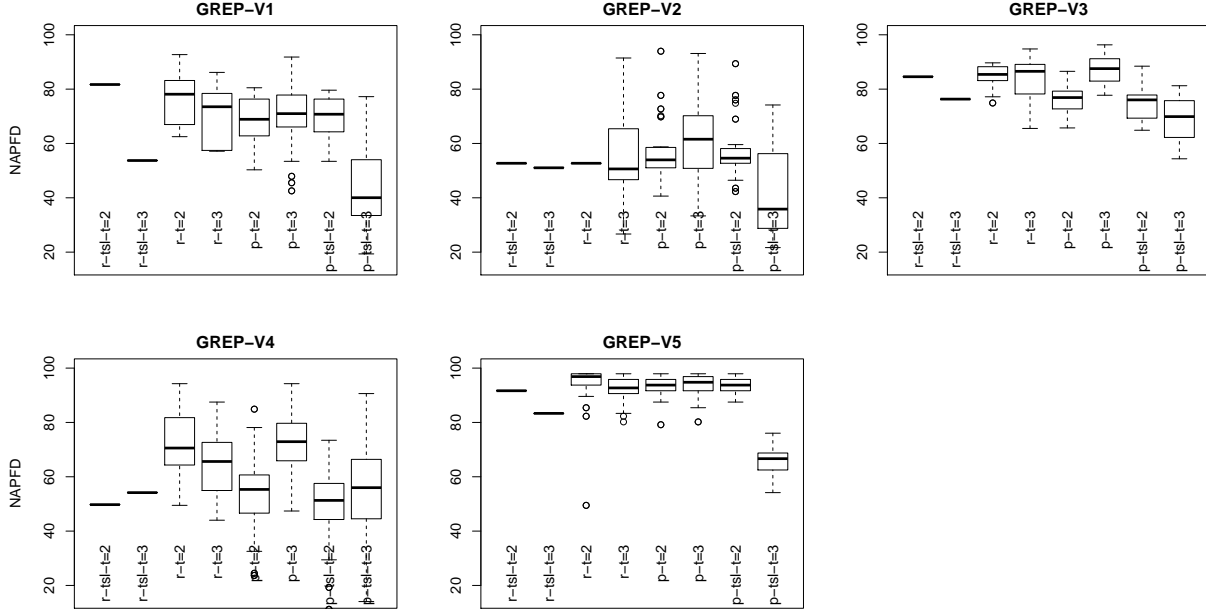


Figure 6. NAPFD for `grep` of different prioritization strength

that **p-t=3** outperforms **p-t=2**, and **r-t=2** exhibits similar cumulative fault detection to **r-t=3**.

In general, in the TSL based weighting, the 3-way techniques do not outperform pair-wise techniques, either for prioritization or regeneration (the only exception is **p-tsl-t=3** and **p-tsl-t=2** in `make`). It may due to how we set weights to the specification.

#### D. Analysis and Discussion

In this section, we first analyze the factors that may impact the results of 3-way prioritization vs. 2-way prioritization, we then provide some suggestions to practitioners, and finally we provide some insights in how we can improve in the future.

First, we believe that the additional benefits of 3-way is dependent on the percentage of 3-way interaction related faults. If we examine the three subject programs in our study (fault detection results in [8]), among all the faults that can be detected by an exhaustive CIT test suite, the 2-way interaction faults make up over 80% of the faults, while 3-way faults only contribute to about 15% of the total. This indicates that if the test suites are generated based on 3-way interactions, bias may be placed on certain 2-way interactions, and other 2-way interaction faults may be missed in early test cases. In general the biased covering arrays are larger than the prioritized arrays meaning that we may miss important pairs early on. We believe that this may explain why the 3-way regenerated suites detect faults slower than the 2-way suites in the beginning.

Second, the poor performance of 3-way TSL based techniques may be due to how we set weights for the specification. We may need different weighting methods for different

testing strengths. We are going to address this in future work.

We are unable to predict the type of faults (i.e., the strength of interactions that causes the fault) before we prioritize and test, therefore, we give suggestions to the practitioners based on previous studies of faults ([8], [9], [21], [22]). Since the literature shows that the 2-way interaction related faults are of higher prevalence than higher strength interaction faults: when code coverage is available, 3-way prioritization (**p-t=3**) may be preferred; if code coverage is unavailable, pair-wise TSL based regeneration (**r-tsl-t=2**) may be the best choice.

Finally, we think there may be some ways to improve the performance of 3-way prioritization. In the current algorithms (Algorithms 1 and 2), the tie breaker of selection is random selection. In the future, when we encounter a tie of 3-way interaction weights, we may break this by considering 2-way interaction weights; i.e. the factor/value/test with higher 2-way interaction weight will be selected.

## VI. CONCLUSIONS

In this paper we have studied several CIT oriented prioritization techniques, with two different testing strengths. Our findings suggest that the prioritization with higher testing strength may lead to earlier fault detection, but the prioritization techniques and weighting method must be considered.

Specifically, if given coverage based weighting, 3-way prioritization (**p-t=3**) is always among the best set of techniques, and it is more effective than pair-wise prioritization (**p-t=2**), in all versions across all three subject programs. However, the result of regeneration is slightly different. 3-way regeneration (**r-t=3**) does not always perform better than pair-wise regeneration (**r-t=2**). It may be due to bias placed

on certain 2-way interactions. If TSL based weighting is considered, we find no 3-way technique that outperforms pair-wise technique, either for prioritization or regeneration. But  $r\text{-tsl-t=2}$  is always among the best techniques.

Therefore, if code coverage is available, 3-way coverage based prioritization should be considered. If we can afford to spend a bit more time, then 3-way regeneration is also effective, but it may not be as good as 2-way in the first few test cases. When we lack code coverage pair-wise TSL based regeneration should be our first choice.

In future work, we are going to analyze the study results from other perspectives, such as *Tuple Density* [17]. We are also going to understand why the TSL based weighting fails to meet our expectation for higher strength prioritization techniques. We will examine different weighting methods for different testing strengths. We are also going to try a smarter tie breaker for  $t$ -way ( $t > 2$ ) prioritization techniques.

#### REFERENCES

- [1] G. Rothermel and M. J. Harrold, "Analyzing regression test selection techniques," *IEEE Transactions on Software Engineering*, vol. 22, no. 8, pp. 529–551, Aug. 1996.
- [2] G. Rothermel, R. Untch, C. Chu, and M. J. Harrold, "Prioritizing test cases for regression testing," *IEEE Transactions on Software Engineering*, vol. 27, no. 10, pp. 929–948, Oct. 2001.
- [3] R. Brownlie, J. Prowse, and M. S. Phadke, "Robust testing of AT&T PMX/StarMAIL using OATS," *AT&T Technical Journal*, vol. 71, no. 3, pp. 41–47, 1992.
- [4] D. M. Cohen, S. R. Dalal, M. L. Fredman, and G. C. Patton, "The AETG system: an approach to testing based on combinatorial design," *IEEE Transactions on Software Engineering*, vol. 23, no. 7, pp. 437–444, 1997.
- [5] M. B. Cohen, C. J. Colbourn, P. B. Gibbons, and W. B. Mugridge, "Constructing test suites for interaction testing," in *Proceedings of the International Conference on Software Engineering*, May 2003, pp. 38–48.
- [6] C. Yilmaz, M. B. Cohen, and A. Porter, "Covering arrays for efficient fault characterization in complex configuration spaces," *IEEE Transactions on Software Engineering*, vol. 31, no. 1, pp. 20–34, Jan 2006.
- [7] S. R. Dalal, A. Jain, N. Karunanithi, J. M. Leaton, C. M. Lott, G. C. Patton, and B. M. Horowitz, "Model-based testing in practice," in *Proceedings of the International Conference on Software Engineering*, 1999, pp. 285–294.
- [8] X. Qu, M. B. Cohen, and K. M. Woolf, "Combinatorial interaction regression testing: A study of test case generation and prioritization," in *Proceedings of the International Conference on Software Maintenance*, 2007, pp. 255–264.
- [9] X. Qu, M. B. Cohen, and G. Rothermel, "Configuration-aware regression testing: an empirical study of sampling and prioritization," in *Proceedings of the International Symposium On Software Testing and Analysis*, 2008, pp. 75–86.
- [10] S. Elbaum, A. G. Malishevsky, and G. Rothermel, "Test case prioritization: A family of empirical studies," *IEEE Transactions on Software Engineering*, vol. 28, no. 2, pp. 159–182, February 2002.
- [11] Z. Li, M. Harman, and R. M. Hierons, "Search algorithms for regression test case prioritization," *IEEE Transactions on Software Engineering*, vol. 33, no. 4, pp. 225–237, April 2007.
- [12] K. R. Walcott, M. L. Soffa, G. M. Kapfhammer, and R. S. Roos, "Time-aware test suite prioritization," in *International Symposium on Software Testing and Analysis*, 2006, pp. 1–11.
- [13] R. Bryce and C. Colbourn, "Prioritized interaction testing for pair-wise coverage with seeding and constraints," *Journal of Information and Software Technology*, vol. 48, no. 10, pp. 960–970, 2006.
- [14] T. J. Ostrand and M. J. Balcer, "The category-partition method for specifying and generating functional tests," *Communications of the ACM*, vol. 31, pp. 678–686, 1988.
- [15] S. Sampath, R. C. Bryce, S. Jain, and S. Manchester, "A tool for combination-based prioritization and reduction of user-session-based test suites," in *Proceedings of the International Conference on Software Maintenance*, ser. ICSM '11, 2011, pp. 574–577.
- [16] S. Manchester, "Combinatorial-based prioritization for user-session-based test suites," Master's thesis, Utah State University, Logan, Utah, 2012.
- [17] B. Chen and J. Zhang, "Tuple density: a new metric for combinatorial test suites (nier track)," in *Proceedings of the 33rd International Conference on Software Engineering*, 2011, pp. 876–879.
- [18] H. Do, S. G. Elbaum, and G. Rothermel, "Supporting controlled experimentation with testing techniques: An infrastructure and its potential impact," *Empirical Software Engineering: An International Journal*, vol. 10, no. 4, pp. 405–435, 2005.
- [19] J. H. Andrews, L. C. Briand, Y. Labiche, and A. S. Namin, "Using mutation analysis for assessing and comparing testing coverage criteria," *IEEE Transactions on Software Engineering*, vol. 32, no. 8, pp. 608–624, 2006.
- [20] M. J. Harrold, L. Larsen, J. Lloyd, D. Nedved, M. Page, and G. Rothermel, "Aristotle: a system for development of program analysis based tools," in *ACM 33rd Annual Southeast Conference*, 1995, pp. 110–119.
- [21] R. Kuhn, R. Kacker, Y. Lei, and J. Hunter, "Combinatorial software testing," *IEEE Computer*, vol. 42, no. 8, pp. 94–96, 2009.
- [22] D. Kuhn, D. R. Wallace, and A. M. Gallo, "Software fault interactions and implications for software testing," *IEEE Transactions on Software Engineering*, vol. 30, no. 6, pp. 418–421, 2004.